

(old)

htmldiff from-

(new)

## Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Copyright © 20222021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12 to suppress diffs in 2022\_03\_RC1 ; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

## AArch32 -- Base Instructions (alphabetic order)

ADC, ADCS (immediate): Add with Carry (immediate).

ADC, ADCS (register): Add with Carry (register).

ADC, ADCS (register-shifted register): Add with Carry (register-shifted register).

ADD (immediate, to PC): Add to PC: an alias of ADR.

ADD, ADDS (immediate): Add (immediate).

ADD, ADDS (register): Add (register).

ADD, ADDS (register-shifted register): Add (register-shifted register).

ADD, ADDS (SP plus immediate): Add to SP (immediate).

ADD, ADDS (SP plus register): Add to SP (register).

ADR: Form PC-relative address.

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

AND, ANDS (register): Bitwise AND (register).

AND, ANDS (register-shifted register): Bitwise AND (register-shifted register).

ASR (immediate): Arithmetic Shift Right (immediate): an alias of MOV, MOVS (register).

ASR (register): Arithmetic Shift Right (register): an alias of MOV, MOVS (register-shifted register).

ASRS (immediate): Arithmetic Shift Right, setting flags (immediate): an alias of MOV, MOVS (register).

ASRS (register): Arithmetic Shift Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

B: Branch.

[BFC](#): Bit Field Clear.

[BFI](#): Bit Field Insert.

BIC, BICS (immediate): Bitwise Bit Clear (immediate).

BIC, BICS (register): Bitwise Bit Clear (register).

BIC, BICS (register-shifted register): Bitwise Bit Clear (register-shifted register).

BKPT: Breakpoint.

BL, BLX (immediate): Branch with Link and optional Exchange (immediate).

BLX (register): Branch with Link and Exchange (register).

BX: Branch and Exchange.

BXJ: Branch and Exchange, previously Branch and Exchange Jazelle.

CBNZ, CBZ: Compare and Branch on Nonzero or Zero.

CLREX: Clear-Exclusive.

CLZ: Count Leading Zeros.

CMN (immediate): Compare Negative (immediate).

CMN (register): Compare Negative (register).

CMN (register-shifted register): Compare Negative (register-shifted register).

CMP (immediate): Compare (immediate).

CMP (register): Compare (register).

CMP (register-shifted register): Compare (register-shifted register).

CPS, CPSID, CPSIE: Change PE State.

CRC32: CRC32.

CRC32C: CRC32C.

CSDB: Consumption of Speculative Data Barrier.

DBG: Debug hint.

DCPS1: Debug Change PE State to EL1.

[DCPS2](#): Debug Change PE State to EL2.

[DCPS3](#): Debug Change PE State to EL3.

DMB: Data Memory Barrier.

[DSB](#): Data Synchronization Barrier.

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

EOR, EORS (register): Bitwise Exclusive OR (register).

EOR, EORS (register-shifted register): Bitwise Exclusive OR (register-shifted register).

ERET: Exception Return.

ESB: Error Synchronization Barrier.

[HLT](#): Halting Breakpoint.

[HVC](#): Hypervisor Call.

ISB: Instruction Synchronization Barrier.

IT: If-Then.

LDA: Load-Acquire Word.

LDAB: Load-Acquire Byte.

LDAEX: Load-Acquire Exclusive Word.

LDAEXB: Load-Acquire Exclusive Byte.

LDAEXD: Load-Acquire Exclusive Doubleword.

LDAEXH: Load-Acquire Exclusive Halfword.

LDAH: Load-Acquire Halfword.

LDC (immediate): Load data to System register (immediate).

LDC (literal): Load data to System register (literal).

LDM (exception return): Load Multiple (exception return).

LDM (User registers): Load Multiple (User registers).

LDM, LDMIA, LDMFD: Load Multiple (Increment After, Full Descending).

LDMDA, LDMFA: Load Multiple Decrement After (Full Ascending).

LDMDB, LDMEA: Load Multiple Decrement Before (Empty Ascending).

LDMIB, LDMED: Load Multiple Increment Before (Empty Descending).

LDR (immediate): Load Register (immediate).

LDR (literal): Load Register (literal).

LDR (register): Load Register (register).

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

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

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

LDRBT: Load Register Byte Unprivileged.

LDRD (immediate): Load Register Dual (immediate).

LDRD (literal): Load Register Dual (literal).

LDRD (register): Load Register Dual (register).

LDREX: Load Register Exclusive.

LDREXB: Load Register Exclusive Byte.

LDREXD: Load Register Exclusive Doubleword.

LDREXH: Load Register Exclusive Halfword.

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

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

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

LDRHT: Load Register Halfword Unprivileged.

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

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

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

LDRSBT: Load Register Signed Byte Unprivileged.

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

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

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

LDRSHT: Load Register Signed Halfword Unprivileged.

LDRT: Load Register Unprivileged.

LSL (immediate): Logical Shift Left (immediate): an alias of MOV, MOVS (register).

LSL (register): Logical Shift Left (register): an alias of MOV, MOVS (register-shifted register).

LSLS (immediate): Logical Shift Left, setting flags (immediate): an alias of MOV, MOVS (register).

LSLS (register): Logical Shift Left, setting flags (register): an alias of MOV, MOVS (register-shifted register).

LSR (immediate): Logical Shift Right (immediate): an alias of MOV, MOVS (register).

LSR (register): Logical Shift Right (register): an alias of MOV, MOVS (register-shifted register).

LSRS (immediate): Logical Shift Right, setting flags (immediate): an alias of MOV, MOVS (register).

LSRS (register): Logical Shift Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

MCR: Move to System register from general-purpose register or execute a System instruction.

MCRR: Move to System register from two general-purpose registers.

MLA, MLAS: Multiply Accumulate.

MLS: Multiply and Subtract.

MOV, MOVS (immediate): Move (immediate).

MOV, MOVS (register): Move (register).

MOV, MOVS (register-shifted register): Move (register-shifted register).

MOVT: Move Top.

MRC: Move to general-purpose register from System register.

MRRC: Move to two general-purpose registers from System register.

MRS: Move Special register to general-purpose register.

MRS (Banked register): Move Banked or Special register to general-purpose register.

MSR (Banked register): Move general-purpose register to Banked or Special register.

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

MSR (register): Move general-purpose register to Special register.

MUL, MULS: Multiply.

MVN, MVNS (immediate): Bitwise NOT (immediate).

MVN, MVNS (register): Bitwise NOT (register).

MVN, MVNS (register-shifted register): Bitwise NOT (register-shifted register).

NOP: No Operation.

ORN, ORNS (immediate): Bitwise OR NOT (immediate).

ORN, ORNS (register): Bitwise OR NOT (register).

ORR, ORRS (immediate): Bitwise OR (immediate).

ORR, ORRS (register): Bitwise OR (register).

ORR, ORRS (register-shifted register): Bitwise OR (register-shifted register).

PKHBT, PKHTB: Pack Halfword.

PLD (literal): Preload Data (literal).

PLD, PLDW (immediate): Preload Data (immediate).

PLD, PLDW (register): Preload Data (register).

PLI (immediate, literal): Preload Instruction (immediate, literal).

PLI (register): Preload Instruction (register).

POP: Pop Multiple Registers from Stack.

POP (multiple registers): Pop Multiple Registers from Stack: an alias of LDM, LDMIA, LDMFD.

POP (single register): Pop Single Register from Stack: an alias of LDR (immediate).

PSSBB: Physical Speculative Store Bypass Barrier.

PUSH: Push Multiple Registers to Stack.

PUSH (multiple registers): Push multiple registers to Stack: an alias of STMDB, STMFD.

PUSH (single register): Push Single Register to Stack: an alias of STR (immediate).

QADD: Saturating Add.

QADD16: Saturating Add 16.

QADD8: Saturating Add 8.

QASX: Saturating Add and Subtract with Exchange.

QDADD: Saturating Double and Add.

QDSUB: Saturating Double and Subtract.

QSAX: Saturating Subtract and Add with Exchange.

QSUB: Saturating Subtract.

QSUB16: Saturating Subtract 16.

QSUB8: Saturating Subtract 8.

RBIT: Reverse Bits.

REV: Byte-Reverse Word.

REV16: Byte-Reverse Packed Halfword.

REVSH: Byte-Reverse Signed Halfword.

RFE, RFEDA, RFEDB, RFEIA, RFEIB: Return From Exception.

ROR (immediate): Rotate Right (immediate): an alias of MOV, MOVS (register).

ROR (register): Rotate Right (register): an alias of MOV, MOVS (register-shifted register).

RORS (immediate): Rotate Right, setting flags (immediate): an alias of MOV, MOVS (register).

RORS (register): Rotate Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

RRX: Rotate Right with Extend: an alias of MOV, MOVS (register).

RRXS: Rotate Right with Extend, setting flags: an alias of MOV, MOVS (register).

RSB, RSBS (immediate): Reverse Subtract (immediate).

RSB, RSBS (register): Reverse Subtract (register).

RSB, RSBS (register-shifted register): Reverse Subtract (register-shifted register).

RSC, RSCS (immediate): Reverse Subtract with Carry (immediate).

RSC, RSCS (register): Reverse Subtract with Carry (register).

RSC, RSCS (register-shifted register): Reverse Subtract (register-shifted register).

SADD16: Signed Add 16.

SADD8: Signed Add 8.

SASX: Signed Add and Subtract with Exchange.

[SB](#): Speculation Barrier.

SBC, SBCS (immediate): Subtract with Carry (immediate).

SBC, SBCS (register): Subtract with Carry (register).

SBC, SBCS (register-shifted register): Subtract with Carry (register-shifted register).

[SBFX](#): Signed Bit Field Extract.

SDIV: Signed Divide.

SEL: Select Bytes.

SETEND: Set Endianness.

SETPAN: Set Privileged Access Never.

SEV: Send Event.

SEVL: Send Event Local.

SHADD16: Signed Halving Add 16.

SHADD8: Signed Halving Add 8.

SHASX: Signed Halving Add and Subtract with Exchange.

SHSAX: Signed Halving Subtract and Add with Exchange.

SHSUB16: Signed Halving Subtract 16.

SHSUB8: Signed Halving Subtract 8.

[SMC](#): Secure Monitor Call.

SMLABB, SMLABT, SMLATB, SMLATT: Signed Multiply Accumulate (halfwords).

SMLAD, SMLADX: Signed Multiply Accumulate Dual.

SMLAL, SMLALS: Signed Multiply Accumulate Long.

SMLALBB, SMLALBT, SMLALTB, SMLALTT: Signed Multiply Accumulate Long (halfwords).

SMLALD, SMLALDX: Signed Multiply Accumulate Long Dual.

SMLAWB, SMLAWT: Signed Multiply Accumulate (word by halfword).

SMLSD, SMLSDX: Signed Multiply Subtract Dual.

SMLSLD, SMLSLDX: Signed Multiply Subtract Long Dual.

SMMLA, SMMLAR: Signed Most Significant Word Multiply Accumulate.

SMMLS, SMMLSR: Signed Most Significant Word Multiply Subtract.

SMMUL, SMMULR: Signed Most Significant Word Multiply.

SMUAD, SMUADX: Signed Dual Multiply Add.

SMULBB, SMULBT, SMULTB, SMULTT: Signed Multiply (halfwords).

SMULL, SMULLS: Signed Multiply Long.

SMULWB, SMULWT: Signed Multiply (word by halfword).

SMUSD, SMUSDX: Signed Multiply Subtract Dual.

[SRS](#), [SRSDA](#), [SRSDB](#), [SRSIA](#), [SRSIB](#): Store Return State.

SSAT: Signed Saturate.

SSAT16: Signed Saturate 16.

SSAX: Signed Subtract and Add with Exchange.

SSBB: Speculative Store Bypass Barrier.

SSUB16: Signed Subtract 16.

SSUB8: Signed Subtract 8.

STC: Store data to System register.

STL: Store-Release Word.

STLB: Store-Release Byte.

STLEX: Store-Release Exclusive Word.

STLEXB: Store-Release Exclusive Byte.

STLEXD: Store-Release Exclusive Doubleword.

STLEXH: Store-Release Exclusive Halfword.

STLH: Store-Release Halfword.

STM (User registers): Store Multiple (User registers).

STM, STMIA, STMEA: Store Multiple (Increment After, Empty Ascending).

STMDA, STMED: Store Multiple Decrement After (Empty Descending).

STMDB, STMFD: Store Multiple Decrement Before (Full Descending).

STMIB, STMFA: Store Multiple Increment Before (Full Ascending).

STR (immediate): Store Register (immediate).

STR (register): Store Register (register).

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

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

STRBT: Store Register Byte Unprivileged.

STRD (immediate): Store Register Dual (immediate).

STRD (register): Store Register Dual (register).

STREX: Store Register Exclusive.

STREXB: Store Register Exclusive Byte.

STREXD: Store Register Exclusive Doubleword.

STREXH: Store Register Exclusive Halfword.

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

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

STRHT: Store Register Halfword Unprivileged.

STRT: Store Register Unprivileged.

SUB (immediate, from PC): Subtract from PC: an alias of ADR.

SUB, SUBS (immediate): Subtract (immediate).

SUB, SUBS (register): Subtract (register).

SUB, SUBS (register-shifted register): Subtract (register-shifted register).

SUB, SUBS (SP minus immediate): Subtract from SP (immediate).

SUB, SUBS (SP minus register): Subtract from SP (register).

SVC: Supervisor Call.

SXTAB: Signed Extend and Add Byte.

SXTAB16: Signed Extend and Add Byte 16.

SXTAH: Signed Extend and Add Halfword.

SXTB: Signed Extend Byte.

SXTB16: Signed Extend Byte 16.

SXTH: Signed Extend Halfword.

TBB, TBH: Table Branch Byte or Halfword.

TEQ (immediate): Test Equivalence (immediate).

TEQ (register): Test Equivalence (register).

TEQ (register-shifted register): Test Equivalence (register-shifted register).

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

TST (immediate): Test (immediate).

TST (register): Test (register).

TST (register-shifted register): Test (register-shifted register).

UADD16: Unsigned Add 16.

UADD8: Unsigned Add 8.

UASX: Unsigned Add and Subtract with Exchange.

[UBFX](#): Unsigned Bit Field Extract.

UDF: Permanently Undefined.

UDIV: Unsigned Divide.

UHADD16: Unsigned Halving Add 16.

UHADD8: Unsigned Halving Add 8.

UHASX: Unsigned Halving Add and Subtract with Exchange.

UHSAX: Unsigned Halving Subtract and Add with Exchange.

UHSUB16: Unsigned Halving Subtract 16.

UHSUB8: Unsigned Halving Subtract 8.

UMAAL: Unsigned Multiply Accumulate Accumulate Long.

UMLAL, UMLALS: Unsigned Multiply Accumulate Long.

UMULL, UMULLS: Unsigned Multiply Long.

UQADD16: Unsigned Saturating Add 16.

UQADD8: Unsigned Saturating Add 8.

UQASX: Unsigned Saturating Add and Subtract with Exchange.

UQSAX: Unsigned Saturating Subtract and Add with Exchange.

UQSUB16: Unsigned Saturating Subtract 16.

UQSUB8: Unsigned Saturating Subtract 8.

USAD8: Unsigned Sum of Absolute Differences.

USADA8: Unsigned Sum of Absolute Differences and Accumulate.

USAT: Unsigned Saturate.

USAT16: Unsigned Saturate 16.

USAX: Unsigned Subtract and Add with Exchange.

USUB16: Unsigned Subtract 16.

USUB8: Unsigned Subtract 8.

UXTAB: Unsigned Extend and Add Byte.

UXTAB16: Unsigned Extend and Add Byte 16.

UXTAH: Unsigned Extend and Add Halfword.

UXTB: Unsigned Extend Byte.

UXTB16: Unsigned Extend Byte 16.

UXTH: Unsigned Extend Halfword.

WFE: Wait For Event.

WFI: Wait For Interrupt.

YIELD: Yield hint.

Internal version only: isa **v01\_27**~~v01\_26~~, pseudocode **v2022-03\_rel**~~v2021-12 to suppress diffs in 2022-03-RC1~~; Build timestamp: **2022-03-29T10:46:11**~~2022-03-08T10:46:11~~

Copyright © **2010-2022**~~2010-2021~~ Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**(old)**

**htmldiff from-**

**(new)**

## BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ) .

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				0	1	1	1	1	1	0	msb					Rd					lsb					0	0	1	1	1	1	1	1
cond																																	

### A1

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

```
d = UInt(Rd); msbit = UInt(msb); lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
if msbit < lsbit then UNPREDICTABLE; if d == 15 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit < lsbit, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	(0)	1	1	0	1	1	0	1	1	1	1	0	imm3				Rd				imm2				(0)	msb			

### T1

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

```
d = UInt(Rd); msbit = UInt(msb); lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit < lsbit then UNPREDICTABLE; if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit < lsbit, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.

<lsb> For encoding A1: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field.

For encoding T1: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field.

<width> Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', (msbit-lsbit)+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

## CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs in 2022-03-RC1; Build timestamp: 2022-03-29T10:46:11

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	1	1	1	1	1	0	msb				Rd				lsb				0 0 1			!= 1111									
cond																												Rn							

### A1

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd); n = UInt(Rn); msbit = UInt(msb); lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
if msbit < lsbit then UNPREDICTABLE; if d == 15 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit < lsbit, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	!= 1111				0	imm3				Rd				imm2		(0)	msb			
Rn																															

### T1

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd); n = UInt(Rn); msbit = UInt(msb); lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit < lsbit then UNPREDICTABLE; if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit < lsbit, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

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

### Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For encoding A1: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;

```

CONSTRAINED UNPREDICTABLE behavior

If msbit < lsbit, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs in 2022-03-RC1 ; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level. DCPS2 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
  - Secure EL2 is not implemented.
  - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR\_hyp, HSR, SPSR\_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP\_EL2, and ELR\_EL2, ESR\_EL2, SPSR\_EL2, DLR\_EL0 and DSPSR\_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

T1

DCPS2

```
if !HaveEL(EL2) then UNDEFINED;
```

## Operation

```
if !Halted() || !() || EL2EnabledIsSecure() then UNDEFINED;

if ELUsingAArch32(EL2) then
    AArch32.WriteMode(M32_Hyp);
    PSTATE.E = HSCTLR.EE;

    ELR_hyp = bits(32) UNKNOWN;
    HSR = bits(32) UNKNOWN;
    SPSR_hyp = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else // Targeting EL2 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL2);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL2;
    if HavePANExt() && SCTLR_EL2.SPAN == '0' && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UA0 = '0';

    ELR_EL2 = bits(64) UNKNOWN;
    ESR_EL2 = bits(64) UNKNOWN;
    SPSR_EL2 = bits(64) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    // SCTLR_EL2.IESB might be ignored in Debug state.
    if HaveIESB() && SCTLR_EL2.IESB == '1' && !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        SynchronizeErrors();

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs\_in\_2022\_03\_RC1; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception level or to a specific mode at the current Exception level.

DCPS3 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes DCPS3:

- If EL3 is using AArch32, the PE enters Monitor mode and LR\_mon, SPSR\_mon, DLR and DSPSR become UNKNOWN. If DCPS3 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP\_EL3, and ELR\_EL3, ESR\_EL3, SPSR\_EL3, DLR\_EL0 and DSPSR\_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

T1

DCPS3

if !HaveEL(EL3) then UNDEFINED;

## Operation

```
if !Halted() || EDSCR.SDD == '1' then UNDEFINED;

if ELUsingAArch32(EL3) then
    from_secure = CurrentSecurityStateIsSecure() == SS_Secure;
    (.);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        PSTATE.E = SCTLR.EE;

    LR_mon = bits(32) UNKNOWN;
    SPSR_mon = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else
    // Targeting EL3 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL3);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL3;
    if HaveUAOExt() then PSTATE.UAO = '0';

    ELR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(64) UNKNOWN;
    SPSR_EL3 = bits(64) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    sync_errors = HaveIESB() && SCTLR_EL3.IESB == '1';
    if HaveDoubleFaultExt() && SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' then
        sync_errors = TRUE;
    // SCTLR_EL3.IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then SynchronizeErrors();

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12 to suppress diffs in 2022-03\_RC1; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## DSB

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

An AArch32 DSB instruction does not require the completion of any AArch64 TLB maintenance instructions, regardless of the nXS qualifier, appearing in program order before the AArch32 DSB.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	!= 0x00			
																												option			

### A1

DSB{<c>}{<q>} {<option>}

// No additional decoding required

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	!= 0x00			
option																															

### T1

DSB{<c>}{<q>} {<option>}

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

## Assembler Symbols

<c> For encoding A1: see [Standard assembler syntax fields](#). Must be AL or omitted.

For encoding T1: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<option> Specifies an optional limitation on the barrier operation. Values are:

#### SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.

#### ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.

#### LD

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.

#### ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.

**ISHST**

Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.

**ISHLD**

Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.

**NSH**

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.

**NSHST**

Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 0b0110.

**NSHLD**

Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.

**OSH**

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.

**OSHST**

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.

**OSHLD**

Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see [Data Synchronization Barrier \(DSB\)](#). All other encodings of option are reserved, other than the values 0b0000 and 0b0100. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

**Note**

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    boolean nXS;
    if HaveFeatXS() && HaveFeatHCX() then
        nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) &&
            IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1');
    else
        nXS = FALSE;
    MBReqDomain domain;
    MBReqTypes types;
    case option of
        when '0001' domain = MBReqDomain_OuterShareable; types = MBReqTypes_Reads;
        when '0010' domain = MBReqDomain_OuterShareable; types = MBReqTypes_Writes;
        when '0011' domain = MBReqDomain_OuterShareable; types = MBReqTypes_All;
        when '0101' domain = MBReqDomain_Nonshareable; types = MBReqTypes_Reads;
        when '0110' domain = MBReqDomain_Nonshareable; types = MBReqTypes_Writes;
        when '0111' domain = MBReqDomain_Nonshareable; types = MBReqTypes_All;
        when '1001' domain = MBReqDomain_InnerShareable; types = MBReqTypes_Reads;
        when '1010' domain = MBReqDomain_InnerShareable; types = MBReqTypes_Writes;
        when '1011' domain = MBReqDomain_InnerShareable; types = MBReqTypes_All;
        when '1101' domain = MBReqDomain_FullSystem; types = MBReqTypes_Reads;
        when '1110' domain = MBReqDomain_FullSystem; types = MBReqTypes_Writes;
    otherwise
        assert !(option IN {'0x00'});
        domain = if option == '0000' then SEE "SSBB";
        elsif option == '0100' then SEE "PSSBB";
        else domain = MBReqDomain_FullSystem; types = MBReqTypes_All;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
            domain = MBReqDomain_OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
            domain = MBReqDomain_InnerShareable;

    DataSynchronizationBarrier(domain, types, nXS);
```

Internal version only: isa ~~v01\_27~~~~v01\_26~~, pseudocode ~~v2022-03\_rel~~~~v2021-12\_to\_suppress\_diffs\_in\_2022\_03\_RC1~~; Build timestamp: ~~2022-03-29T10:46:11~~~~2022-03-08T10:46:11~~

Copyright © ~~2010-2022~~~~2010-2021~~ Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## HLT

Halting breakpoint causes a software breakpoint to occur.

Halting breakpoint is always unconditional, even inside an IT block.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	0	0	imm12												0	1	1	1	imm4			
cond																															

### A1

HLT{<q>} {#}<imm>

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE; // HLT must be encoded with AL condition
```

## CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	0	imm6					

### T1

HLT{<q>} {#}<imm>

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
```

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

## Assembler Symbols

- <q> See [Standard assembler syntax fields](#). An HLT instruction must be unconditional.
- <imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.
- For encoding T1: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

## Operation

```
EncodingSpecificOperations();
boolean is_async = FALSE;EncodingSpecificOperations();
Halt(DebugHalt_HaltInstruction, is_async);;
```

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12 to suppress diffs in 2022-03-RC1; Build timestamp: 2022-03-29T10:20:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see [Hypervisor Call \(HVC\) exception](#). Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and using AArch64, and [SCR\\_EL3](#).HCE is set to 0.
- In Non-secure EL1 modes when EL3 is implemented and using AArch32, and [SCR](#).HCE is set to 0.
- When EL3 is not implemented and either [HCR\\_EL2](#).HCD is set to 1 or [HCR](#).HCD is set to 1.
- When EL2 is not implemented.
- In Secure state, if EL2 is not enabled in the current Security state.
- In User mode.

The HVC instruction is CONSTRAINED UNPREDICTABLE in Hyp mode when EL3 is implemented and using AArch32, and [SCR](#).HCE is set to 0.

On executing an HVC instruction, the [HSR, Hyp Syndrome Register](#) reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see [Use of the HSR](#).

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	0	0	imm12												0	1	1	1	imm4			
cond																															

### A1

HVC{<q>} {#}<imm16>

```
if cond != '1110' then UNPREDICTABLE;
imm16 = imm12:imm4;
```

### CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	0	imm4				1	0	0	0	imm12											

### T1

HVC{<q>} {#}<imm16>

```
imm16 = imm4:imm12;
if InITBlock() then UNPREDICTABLE;
```

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

### Assembler Symbols

<q> See [Standard assembler syntax fields](#). An HVC instruction must be unconditional.

<imm16> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

## Operation

```

EncodingSpecificOperations();
if PSTATE.EL IN {if !HaveEL(EL2) || PSTATE.EL == EL0, || ( EL3IsSecure } || !() && !EL2EnabledIsSecureEL2En
(()) then
    UNDEFINED;

bit hvc_enable;
if HaveEL(EL3) then
    if ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2 then
        UNPREDICTABLE;
    else
        hvc_enable = SCR_GEN[].HCE;
else
    hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR.HCD) else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch32.CallHypervisor(imm16);

```

## CONSTRAINED UNPREDICTABLE behavior

If ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs in 2022-03\_RC1; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

#### (FEAT\_SB)

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

### A1

SB{<q>}

// No additional decoding required

### T1

#### (FEAT\_SB)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	1	(0)	(0)	(0)	(0)

### T1

SB{<q>}

if [InITBlock\(\)](#) then UNPREDICTABLE;

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

## Assembler Symbols

<q> See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculationBarrier();
```

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs\_in\_2022\_03\_RC1; Build timestamp: 2022-03-29T10:46:112022-03-08T10:46:11

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	0	1	widthm1					Rd				lsb				1 0 1			Rn				
cond																															

### A1

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(lsb); widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE; if d == 15 || n == 15 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	0	0	Rn				0	imm3				Rd				imm2		(0)	widthm1			

### T1

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(imm3:imm2); widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit > 31 then UNPREDICTABLE; if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

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

## Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.  For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;

```

## CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Internal version only: isa [v01\\_27](#)~~v01\_26~~, pseudocode [v2022-03\\_rel](#)~~v2021-12 to suppress diffs in 2022-03-RC1~~; Build timestamp: [2022-03-29T10:46:11](#)~~2022-03-08T10:46:11~~

Copyright © [2010-2022](#)~~2010-2021~~ Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information see [Secure Monitor Call \(SMC\) exception](#).

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of [HCR.TSC](#) and [SCR.SCD](#) are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the [SCR\\_EL3.SMD](#) bit, rather than the [SCR.SCD](#) bit, that can change the effect of executing an SMC instruction.

If the value of [HCR.TSC](#) is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of [SCR.SCD](#). When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information see [Traps to Hyp mode of Non-secure EL1 execution of SMC instructions](#).

If the value of [HCR.TSC](#) is 0 and the value of [SCR.SCD](#) is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ) .

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	1	imm4			
cond																															

### A1

```
SMC{<c>}{<q>} {#}<imm4>
```

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	1	imm4				1	0	0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

### T1

```
SMC{<c>}{<q>} {#}<imm4>
```

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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

## Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<imm4> Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the "imm4" field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    AArch32.CheckForSMCUndefinedOrTrap();

    if !ELUsingAArch32(EL3) then
        if SCR_EL3.SMD == '1' then
            // SMC disabled.
            UNDEFINED;
        else
            if SCR.SCD == '1' then
                // SMC disabled
                if CurrentSecurityStateIsSecure() == SS_Secure then
                    // Executes either as a NOP or UNALLOCATED.
                    c = ConstrainUnpredictable(Unpredictable_SMD);
                    assert c IN {Constraint_NOP, Constraint_UNDEF};
                    if c == Constraint_NOP then EndOfInstruction();
                    UNDEFINED;

            if !ELUsingAArch32(EL3) then
                AArch64.CallSecureMonitor(Zeros(16));
            else
                AArch32.TakeSMCException();
```

## CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && CurrentSecurityStateIsSecure() == SS_Secure`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs in 2022\_03\_RC1; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR\_<current\_mode> and *SPSR*\_<current\_mode> to the stack of a specified mode. For information about memory accesses see *Memory accesses*.

SRS is UNDEFINED in Hyp mode.

SRS is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that *Table G1-5* does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.

If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) and [T2](#) ).

### A1

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

#### Decrement After (P == 0 && U == 0)

SRSDA{<c>}{<q>} SP{!}, #<mode>

#### Decrement Before (P == 1 && U == 0)

SRSDB{<c>}{<q>} SP{!}, #<mode>

#### Increment After (P == 0 && U == 1)

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

#### Increment Before (P == 1 && U == 1)

SRSIB{<c>}{<q>} SP{!}, #<mode>

wback = (W == '1'); increment = (U == '1'); wordhigher = (P == U);

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

### T1

SRSDB{<c>}{<q>} SP{!}, #<mode>

wback = (W == '1'); increment = FALSE; wordhigher = FALSE;

### T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

## T2

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

```
wback = (W == '1'); increment = TRUE; wordhigher = FALSE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [SRS \(T32\)](#) and [SRS \(A32\)](#).

## Assembler Symbols

IA	For encoding A1: is an optional suffix to indicate the Increment After variant. For encoding T2: is an optional suffix for the Increment After form.
<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . <c> must be AL or omitted. For encoding T1 and T2: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<mode>	Is the number of the mode whose Banked SP is used as the base register, encoded in the "mode" field. For details of PE modes and their numbers see <a href="#">AArch32 PE mode descriptions</a> .

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDb, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

## Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then          // UNDEFINED at EL2
      UNDEFINED;

    // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
    // to be security holes
    if PSTATE.M IN {M32_User,M32_System} then
      UNPREDICTABLE;
    elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
      UNPREDICTABLE;
    elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
      if !HaveEL(EL3) || || ! CurrentSecurityStateIsSecure() !=() then
        UNPREDICTABLE;
      elsif ! SS_Secure then
        UNPREDICTABLE;
      elsif !ELUsingAArch32(EL3) then
        AArch64.MonitorModeTrap();
    elsif BadMode(mode) then
      UNPREDICTABLE;

    base = Rmode[13,mode];
    address = if increment then base else base-8;
    if wordhigher then address = address+4;
    MemA[address,4] = LR;
    MemA[address+4,4] = SPSR[];
    if wback then Rmode[13,mode] = if increment then base+8 else base-8;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      if PSTATE.EL == EL2 then          // UNDEFINED at EL2
        UNDEFINED;

      // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
      // to be security holes
      if PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;
      elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
        UNPREDICTABLE;
      elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
        if !HaveEL(EL3) || || ! CurrentSecurityStateIsSecure() != SS_Secure then
          () then
            UNPREDICTABLE;
          elsif !ELUsingAArch32(EL3) then
            AArch64.MonitorModeTrap();
          elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4] = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
```

## CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Hyp`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Monitor && (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure)!!IsSecure()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `BadMode(mode)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction stores to the stack of the mode in which it is executed.
- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.

Internal version only: isa `v01_27v01_26`, pseudocode `v2022-03_relv2021-12 to suppress diffs in 2022_03_RC1`; Build timestamp: `2022-03-29T10:20:46+00:00`

Copyright © `2010-20222010-2021` Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions.

If [FEAT\\_TRF](#) is not implemented, this instruction executes as a NOP.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

#### (FEAT\_TRF)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	0	1	0
cond																															

### A1

TSB{<c>}{<q>} CSYNC

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
```

```
if cond != '1110' then UNPREDICTABLE; // TSB must be encoded with AL condition
```

```
if cond != '1110' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### T1

#### (FEAT\_TRF)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	1	0	0	1	0	0

### T1

TSB{<c>}{<q>} CSYNC

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
```

```
if InITBlock() then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

## Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    TraceSynchronizationBarrier();
```

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs\_in\_2022\_03\_RC1; Build timestamp: 2022-03-29T10:46:112022-03-08T10:46:11

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	1	1	widthm1				Rd				lsb				1 0 1				Rn				
cond																															

### A1

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(lsb); widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE; if d == 15 || n == 15 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	1	0	0	Rn				0	imm3				Rd				imm2		(0)	widthm1			

### T1

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(imm3:imm2); widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit > 31 then UNPREDICTABLE; if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE
```

### CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

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

## Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.  For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;

```

## CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Internal version only: isa [v01\\_27](#)~~v01\_26~~, pseudocode [v2022-03\\_rel](#)~~v2021-12 to suppress diffs in 2022-03-RC1~~; Build timestamp: [2022-03-29T10:46:11](#)~~2022-03-08T10:46:11~~

Copyright © [2010-2022](#)~~2010-2021~~ Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

## AArch32 -- SIMD&FP Instructions (alphabetic order)

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

FLDM\*X (FLDMDBX, FLDMIAX): FLDM\*X.

FSTMDBX, FSTMIAX: FSTMX.

SHA1C: SHA1 hash update (choose).

SHA1H: SHA1 fixed rotate.

SHA1M: SHA1 hash update (majority).

SHA1P: SHA1 hash update (parity).

SHA1SU0: SHA1 schedule update 0.

SHA1SU1: SHA1 schedule update 1.

SHA256H: SHA256 hash update part 1.

SHA256H2: SHA256 hash update part 2.

SHA256SU0: SHA256 schedule update 0.

SHA256SU1: SHA256 schedule update 1.

VABA: Vector Absolute Difference and Accumulate.

VABAL: Vector Absolute Difference and Accumulate Long.

VABD (floating-point): Vector Absolute Difference (floating-point).

VABD (integer): Vector Absolute Difference (integer).

VABDL (integer): Vector Absolute Difference Long (integer).

VABS: Vector Absolute.

VACGE: Vector Absolute Compare Greater Than or Equal.

VACGT: Vector Absolute Compare Greater Than.

VACLE: Vector Absolute Compare Less Than or Equal: an alias of VACGE.

VACLT: Vector Absolute Compare Less Than: an alias of VACGT.

VADD (floating-point): Vector Add (floating-point).

VADD (integer): Vector Add (integer).

VADDHN: Vector Add and Narrow, returning High Half.

VADDL: Vector Add Long.

VADDW: Vector Add Wide.

VAND (immediate): Vector Bitwise AND (immediate): an alias of VBIC (immediate).

VAND (register): Vector Bitwise AND (register).

VBIC (immediate): Vector Bitwise Bit Clear (immediate).

VBIC (register): Vector Bitwise Bit Clear (register).

VBIF: Vector Bitwise Insert if False.

VBIT: Vector Bitwise Insert if True.

VBSL: Vector Bitwise Select.

VCADD: Vector Complex Add.

VCEQ (immediate #0): Vector Compare Equal to Zero.

VCEQ (register): Vector Compare Equal.

VCGE (immediate #0): Vector Compare Greater Than or Equal to Zero.

VCGE (register): Vector Compare Greater Than or Equal.

VCGT (immediate #0): Vector Compare Greater Than Zero.

VCGT (register): Vector Compare Greater Than.

VCLE (immediate #0): Vector Compare Less Than or Equal to Zero.

VCLE (register): Vector Compare Less Than or Equal: an alias of VCGE (register).

VCLS: Vector Count Leading Sign Bits.

VCLT (immediate #0): Vector Compare Less Than Zero.

VCLT (register): Vector Compare Less Than: an alias of VCGT (register).

VCLZ: Vector Count Leading Zeros.

VCMLA: Vector Complex Multiply Accumulate.

VCMLA (by element): Vector Complex Multiply Accumulate (by element).

VCMP: Vector Compare.

VCMPPE: Vector Compare, raising Invalid Operation on NaN.

VCNT: Vector Count Set Bits.

VCVT (between double-precision and single-precision): Convert between double-precision and single-precision.

VCVT (between floating-point and fixed-point, Advanced SIMD): Vector Convert between floating-point and fixed-point.

VCVT (between floating-point and fixed-point, floating-point): Convert between floating-point and fixed-point.

VCVT (between floating-point and integer, Advanced SIMD): Vector Convert between floating-point and integer.

VCVT (between half-precision and single-precision, Advanced SIMD): Vector Convert between half-precision and single-precision.

VCVT (floating-point to integer, floating-point): Convert floating-point to integer with Round towards Zero.

VCVT (from single-precision to BFloat16, Advanced SIMD): Vector Convert from single-precision to BFloat16.

VCVT (integer to floating-point, floating-point): Convert integer to floating-point.

VCVTA (Advanced SIMD): Vector Convert floating-point to integer with Round to Nearest with Ties to Away.

VCVTA (floating-point): Convert floating-point to integer with Round to Nearest with Ties to Away.

VCVTB: Convert to or from a half-precision value in the bottom half of a single-precision register.

VCVTB (BFloat16): Converts from a single-precision value to a BFloat16 value in the bottom half of a single-precision register.

VCVTM (Advanced SIMD): Vector Convert floating-point to integer with Round towards -Infinity.

VCVTM (floating-point): Convert floating-point to integer with Round towards -Infinity.

VCVTN (Advanced SIMD): Vector Convert floating-point to integer with Round to Nearest.

VCVTN (floating-point): Convert floating-point to integer with Round to Nearest.

VCVTP (Advanced SIMD): Vector Convert floating-point to integer with Round towards +Infinity.

VCVTP (floating-point): Convert floating-point to integer with Round towards +Infinity.

VCVTR: Convert floating-point to integer.

VCVTT: Convert to or from a half-precision value in the top half of a single-precision register.

VCVTT (BFloat16): Converts from a single-precision value to a BFloat16 value in the top half of a single-precision register..

VDIV: Divide.

VDOT (by element): BFloat16 floating-point indexed dot product (vector, by element).

VDOT (vector): BFloat16 floating-point (BF16) dot product (vector).

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

VDUP (scalar): Duplicate vector element to vector.

VEOR: Vector Bitwise Exclusive OR.

VEXT (byte elements): Vector Extract.

VEXT (multibyte elements): Vector Extract: an alias of VEXT (byte elements).

VFMA: Vector Fused Multiply Accumulate.

VFMA, VFMA, VFMA (BFloat16, by scalar): BFloat16 floating-point widening multiply-add long (by scalar).

VFMA, VFMA, VFMA (BFloat16, vector): BFloat16 floating-point widening multiply-add long (vector).

VFMA (by scalar): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

VFMA (vector): Vector Floating-point Multiply-Add Long to accumulator (vector).

VFMS: Vector Fused Multiply Subtract.

VFMSL (by scalar): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

VFMSL (vector): Vector Floating-point Multiply-Subtract Long from accumulator (vector).

VFNMA: Vector Fused Negate Multiply Accumulate.

VFNMS: Vector Fused Negate Multiply Subtract.

VHADD: Vector Halving Add.

VHSUB: Vector Halving Subtract.

VINS: Vector move Insertion.

VJCVT: Javascript Convert to signed fixed-point, rounding toward Zero.

VLD1 (multiple single elements): Load multiple single 1-element structures to one, two, three, or four registers.

VLD1 (single element to all lanes): Load single 1-element structure and replicate to all lanes of one register.

VLD1 (single element to one lane): Load single 1-element structure to one lane of one register.

VLD2 (multiple 2-element structures): Load multiple 2-element structures to two or four registers.

VLD2 (single 2-element structure to all lanes): Load single 2-element structure and replicate to all lanes of two registers.

VLD2 (single 2-element structure to one lane): Load single 2-element structure to one lane of two registers.

VLD3 (multiple 3-element structures): Load multiple 3-element structures to three registers.

VLD3 (single 3-element structure to all lanes): Load single 3-element structure and replicate to all lanes of three registers.

VLD3 (single 3-element structure to one lane): Load single 3-element structure to one lane of three registers.

VLD4 (multiple 4-element structures): Load multiple 4-element structures to four registers.

VLD4 (single 4-element structure to all lanes): Load single 4-element structure and replicate to all lanes of four registers.

VLD4 (single 4-element structure to one lane): Load single 4-element structure to one lane of four registers.

VLDM, VLDMDB, VLDMIA: Load Multiple SIMD&FP registers.

VLDR (immediate): Load SIMD&FP register (immediate).

VLDR (literal): Load SIMD&FP register (literal).

VMAX (floating-point): Vector Maximum (floating-point).

VMAX (integer): Vector Maximum (integer).

VMAXNM: Floating-point Maximum Number.

VMIN (floating-point): Vector Minimum (floating-point).

VMIN (integer): Vector Minimum (integer).

VMINNM: Floating-point Minimum Number.

VMLA (by scalar): Vector Multiply Accumulate (by scalar).

VMLA (floating-point): Vector Multiply Accumulate (floating-point).

VMLA (integer): Vector Multiply Accumulate (integer).

VMLAL (by scalar): Vector Multiply Accumulate Long (by scalar).

VMLAL (integer): Vector Multiply Accumulate Long (integer).

VMLS (by scalar): Vector Multiply Subtract (by scalar).

VMLS (floating-point): Vector Multiply Subtract (floating-point).

VMLS (integer): Vector Multiply Subtract (integer).

VMLSL (by scalar): Vector Multiply Subtract Long (by scalar).

VMLSL (integer): Vector Multiply Subtract Long (integer).

VMMLA: BFloat16 floating-point matrix multiply-accumulate.

VMOV (between general-purpose register and half-precision): Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register.

VMOV (between general-purpose register and single-precision): Copy a general-purpose register to or from a 32-bit SIMD&FP register.

VMOV (between two general-purpose registers and a doubleword floating-point register): Copy two general-purpose registers to or from a SIMD&FP register.

VMOV (between two general-purpose registers and two single-precision registers): Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers.

VMOV (general-purpose register to scalar): Copy a general-purpose register to a vector element.

VMOV (immediate): Copy immediate value to a SIMD&FP register.

VMOV (register): Copy between FP registers.

VMOV (register, SIMD): Copy between SIMD registers: an alias of VORR (register).

VMOV (scalar to general-purpose register): Copy a vector element to a general-purpose register with sign or zero extension.

VMOVL: Vector Move Long.

VMOVN: Vector Move and Narrow.

VMOVX: Vector Move extraction.

VMRS: Move SIMD&FP Special register to general-purpose register.

VMSR: Move general-purpose register to SIMD&FP Special register.

VMUL (by scalar): Vector Multiply (by scalar).

VMUL (floating-point): Vector Multiply (floating-point).

VMUL (integer and polynomial): Vector Multiply (integer and polynomial).

VMULL (by scalar): Vector Multiply Long (by scalar).

VMULL (integer and polynomial): Vector Multiply Long (integer and polynomial).

VMVN (immediate): Vector Bitwise NOT (immediate).

VMVN (register): Vector Bitwise NOT (register).

VNEG: Vector Negate.

VNMLA: Vector Negate Multiply Accumulate.

VNMLS: Vector Negate Multiply Subtract.

VNMUL: Vector Negate Multiply.

VORN (immediate): Vector Bitwise OR NOT (immediate): an alias of VORR (immediate).

VORN (register): Vector bitwise OR NOT (register).

VORR (immediate): Vector Bitwise OR (immediate).

VORR (register): Vector bitwise OR (register).

VPADAL: Vector Pairwise Add and Accumulate Long.

VPADD (floating-point): Vector Pairwise Add (floating-point).

VPADD (integer): Vector Pairwise Add (integer).

VPADDL: Vector Pairwise Add Long.

VPMAX (floating-point): Vector Pairwise Maximum (floating-point).

VPMAX (integer): Vector Pairwise Maximum (integer).

VPMIN (floating-point): Vector Pairwise Minimum (floating-point).

VPMIN (integer): Vector Pairwise Minimum (integer).

VPOP: Pop SIMD&FP registers from stack: an alias of VLDM, VLDMDB, VLDMIA.

VPUSH: Push SIMD&FP registers to stack: an alias of VSTM, VSTMDB, VSTMIA.

VQABS: Vector Saturating Absolute.

VQADD: Vector Saturating Add.

VQDMLAL: Vector Saturating Doubling Multiply Accumulate Long.

VQDMLSL: Vector Saturating Doubling Multiply Subtract Long.

VQDMULH: Vector Saturating Doubling Multiply Returning High Half.

VQDMULL: Vector Saturating Doubling Multiply Long.

VQMOVN, VQMOVUN: Vector Saturating Move and Narrow.

VQNEG: Vector Saturating Negate.

VQRDMLAH: Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half.

VQRDMLSH: Vector Saturating Rounding Doubling Multiply Subtract Returning High Half.

VQRDMULH: Vector Saturating Rounding Doubling Multiply Returning High Half.

[VQRSHL](#): Vector Saturating Rounding Shift Left.

VQRSHRN (zero): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQRSHRN, VQRSHRUN: Vector Saturating Rounding Shift Right, Narrow.

VQRSHRUN (zero): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSHL (register): Vector Saturating Shift Left (register).

VQSHL, VQSHLU (immediate): Vector Saturating Shift Left (immediate).

VQSHRN (zero): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSHRN, VQSHRUN: Vector Saturating Shift Right, Narrow.

VQSHRUN (zero): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSUB: Vector Saturating Subtract.

VRADDHN: Vector Rounding Add and Narrow, returning High Half.

VRECPE: Vector Reciprocal Estimate.

VRECPS: Vector Reciprocal Step.

VREV16: Vector Reverse in halfwords.

VREV32: Vector Reverse in words.

VREV64: Vector Reverse in doublewords.

VRHADD: Vector Rounding Halving Add.

VRINTA (Advanced SIMD): Vector Round floating-point to integer towards Nearest with Ties to Away.

VRINTA (floating-point): Round floating-point to integer to Nearest with Ties to Away.

VRINTM (Advanced SIMD): Vector Round floating-point to integer towards -Infinity.

VRINTM (floating-point): Round floating-point to integer towards -Infinity.

VRINTN (Advanced SIMD): Vector Round floating-point to integer to Nearest.

VRINTN (floating-point): Round floating-point to integer to Nearest.

VRINTP (Advanced SIMD): Vector Round floating-point to integer towards +Infinity.

VRINTP (floating-point): Round floating-point to integer towards +Infinity.

VRINTR: Round floating-point to integer.

VRINTX (Advanced SIMD): Vector round floating-point to integer inexact.

VRINTX (floating-point): Round floating-point to integer inexact.

VRINTZ (Advanced SIMD): Vector round floating-point to integer towards Zero.

VRINTZ (floating-point): Round floating-point to integer towards Zero.

**VRSHL**: Vector Rounding Shift Left.

VRSHR: Vector Rounding Shift Right.

VRSHR (zero): Vector Rounding Shift Right: an alias of VORR (register).

VRSHRN: Vector Rounding Shift Right and Narrow.

VRSHRN (zero): Vector Rounding Shift Right and Narrow: an alias of VMOVN.

VRSQRTE: Vector Reciprocal Square Root Estimate.

VRSQRTS: Vector Reciprocal Square Root Step.

VRSRA: Vector Rounding Shift Right and Accumulate.

VRSUBHN: Vector Rounding Subtract and Narrow, returning High Half.

VSDOT (by element): Dot Product index form with signed integers..

VSDOT (vector): Dot Product vector form with signed integers..

VSELEQ, VSELGE, VSELGT, VSELVS: Floating-point conditional select.

VSHL (immediate): Vector Shift Left (immediate).

VSHL (register): Vector Shift Left (register).

VSHLL: Vector Shift Left Long.

VSHR: Vector Shift Right.

VSHR (zero): Vector Shift Right: an alias of VORR (register).

VSHRN: Vector Shift Right Narrow.

VSHRN (zero): Vector Shift Right Narrow: an alias of VMOVN.

VSLI: Vector Shift Left and Insert.

VSMMLA: Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix.

VSQRT: Square Root.

VSRA: Vector Shift Right and Accumulate.

VSRI: Vector Shift Right and Insert.

VST1 (multiple single elements): Store multiple single elements from one, two, three, or four registers.

VST1 (single element from one lane): Store single element from one lane of one register.

VST2 (multiple 2-element structures): Store multiple 2-element structures from two or four registers.

VST2 (single 2-element structure from one lane): Store single 2-element structure from one lane of two registers.

VST3 (multiple 3-element structures): Store multiple 3-element structures from three registers.

VST3 (single 3-element structure from one lane): Store single 3-element structure from one lane of three registers.

VST4 (multiple 4-element structures): Store multiple 4-element structures from four registers.

VST4 (single 4-element structure from one lane): Store single 4-element structure from one lane of four registers.

VSTM, VSTMDB, VSTMIA: Store multiple SIMD&FP registers.

VSTR: Store SIMD&FP register.

VSUB (floating-point): Vector Subtract (floating-point).

VSUB (integer): Vector Subtract (integer).

VSUBHN: Vector Subtract and Narrow, returning High Half.

VSUBL: Vector Subtract Long.

VSUBW: Vector Subtract Wide.

VSUDOT (by element): Dot Product index form with signed and unsigned integers (by element).

VSWP: Vector Swap.

VTBL, VTBX: Vector Table Lookup and Extension.

VTRN: Vector Transpose.

VTST: Vector Test Bits.

VUDOT (by element): Dot Product index form with unsigned integers..

VUDOT (vector): Dot Product vector form with unsigned integers..

VUMMLA: Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix.

VUSDOT (by element): Dot Product index form with unsigned and signed integers (by element).

VUSDOT (vector): Dot Product vector form with mixed-sign integers.

VUSMMLA: Widening 8-bit mixed integer matrix multiply-accumulate into 2x2 matrix.

VUZP: Vector Unzip.

VUZP (alias): Vector Unzip: an alias of VTRN.

VZIP: Vector Zip.

VZIP (alias): Vector Zip: an alias of VTRN.

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs in 2022-03-RC1; Build timestamp: 2022-03-29T10:2022-03-08T10:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see [VQSHL \(register\)](#).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs.

For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

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

#### 64-bit SIMD vector (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>

#### 128-bit SIMD vector (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
```

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size	Vn					Vd					0	1	0	1	N	Q	M	1	Vm				

#### 64-bit SIMD vector (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>

#### 128-bit SIMD vector (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
 For encoding T1: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the elements of the vectors, encoded in "U:size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>\*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>\*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(esize) result;
    boolean sat;
    for r = 0 to regs-1
        for e = 0 to elements-1
            integer element = IntSIntIntSInt(Elem[D[m+r], e, esize], unsigned);
            integer shift = [n+r],e,esize<7:0>;
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = SIntInt(Elem[D[n+r], e, esize]<7:0>);
            if shift >= 0 then // left shift
                element = element << shift;
            else // rounding right shift
                shift = -shift;
                element = (element + (1 << (shift - 1))) >> shift;
            [m+r],e,esize], unsigned);
            (result, sat) = SatQ(element, esize, unsigned);((operand + round_const) << shift, esize, unsigned);
            Elem[D[d+r], e, esize] = result;
            [d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa ~~v01\_27~~v01\_26, pseudocode ~~v2022-03\_rel~~v2021-12 to suppress diffs in 2022\_03\_RC1; Build timestamp: ~~2022-03-29T10:46:11~~2022-03-08T10:46:11

Copyright © ~~2010-2022~~2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ( [A1](#) ) and T32 ( [T1](#) ).

### A1

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

#### 64-bit SIMD vector (Q == 0)

VRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>

#### 128-bit SIMD vector (Q == 1)

VRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
```

### T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size	Vn					Vd					0	1	0	1	N	Q	M	0	Vm				

#### 64-bit SIMD vector (Q == 0)

VRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>

#### 128-bit SIMD vector (Q == 1)

VRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
 For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

<dt>	Is the data type for the elements of the vectors, encoded in "U:size":
------	--

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

**<Qd>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.

**<Qm>** Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>\*2.

<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
------	--

**<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
------	---

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();    CheckAdvSIMDEnabled();
    integer result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            integer element =          shift = IntSInt(Elem[D[m+r], e, esize], unsigned);
            integer shift = [n+r],e,esize]<7:0>;
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = ( SIntInt(Elem[D[n+r], e, esize]<7:0>);
            if shift >= 0 then // left shift
                result = element << shift;
            else // rounding right shift
                shift = -shift;
                result = (element + (1 << (shift - 1))) >> shift; [m+r],e,esize], unsigned) + round_const;
            Elem[D[d+r], e, esize] = result<esize-1:0>; [d+r],e,esize] = result<esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs\_in\_2022\_03\_RC1 ; Build timestamp: 2022-03-29T10:20:46-08002022-03-08T10:46-11

Copyright © 2010-20222010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

**(new)**

(old)

htmldiff from-

(new)

## Top-level encodings for A32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				op0																						op1					

Decode fields			Instruction details
cond	op0	op1	
!= 1111	00x		<a href="#">Data-processing and miscellaneous instructions</a>
!= 1111	010		<a href="#">Load/Store Word, Unsigned Byte (immediate, literal)</a>
!= 1111	011	0	<a href="#">Load/Store Word, Unsigned Byte (register)</a>
!= 1111	011	1	<a href="#">Media instructions</a>
	10x		<a href="#">Branch, branch with link, and block data transfer</a>
	11x		<a href="#">System register access, Advanced SIMD, floating-point, and Supervisor call</a>
1111	0xx		<a href="#">Unconditional instructions</a>

### Data-processing and miscellaneous instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				00		op0	op1																				op2	op3	op4		

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0		1	!= 00	1	<a href="#">Extra load/store</a>
0	0xxxx	1	00	1	<a href="#">Multiply and Accumulate</a>
0	1xxxx	1	00	1	<a href="#">Synchronization primitives and Load-Acquire/Store-Release</a>
0	10xx0	0			<a href="#">Miscellaneous</a>
0	10xx0	1		0	<a href="#">Halfword Multiply and Accumulate</a>
0	!= 10xx0			0	<a href="#">Data-processing register (immediate shift)</a>
0	!= 10xx0	0		1	<a href="#">Data-processing register (register shift)</a>
1					<a href="#">Data-processing immediate</a>

### Extra load/store

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				000			op0																		1	!= 00	1				

Decode fields		Instruction details
op0		
0		<a href="#">Load/Store Dual, Half, Signed Byte (register)</a>
1		<a href="#">Load/Store Dual, Half, Signed Byte (immediate, literal)</a>

### Load/Store Dual, Half, Signed Byte (register)

These instructions are under [Extra load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				0	0	0	P	U	0	W	op1	Rn				Rt				(0)		(0)		(0)		(0)		1	!= 00	1	Rm		
cond												op2																					

The following constraints also apply to this encoding: `cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00`

Decode fields				Instruction Details
P	W	o1	op2	
0	0	0	01	STRH (register) — post-indexed
0	0	0	10	LDRD (register) — post-indexed
0	0	0	11	STRD (register) — post-indexed
0	0	1	01	LDRH (register) — post-indexed
0	0	1	10	LDRSB (register) — post-indexed
0	0	1	11	LDRSH (register) — post-indexed
0	1	0	01	STRHT
0	1	0	10	UNALLOCATED
0	1	0	11	UNALLOCATED
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1		0	01	STRH (register) — pre-indexed
1		0	10	LDRD (register) — pre-indexed
1		0	11	STRD (register) — pre-indexed
1		1	01	LDRH (register) — pre-indexed
1		1	10	LDRSB (register) — pre-indexed
1		1	11	LDRSH (register) — pre-indexed

### Load/Store Dual, Half, Signed Byte (immediate, literal)

These instructions are under [Extra load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	o1	Rn				Rt				imm4H				1	!= 00		1	imm4L			
cond												op2																			

The following constraints also apply to this encoding: `cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00`

Decode fields				Instruction Details
P:W	o1	Rn	op2	
	0	1111	10	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) — post-indexed
00	0		01	STRH (immediate) — post-indexed
00	0		11	STRD (immediate) — post-indexed
00	1	!= 1111	01	LDRH (immediate) — post-indexed
00	1	!= 1111	10	LDRSB (immediate) — post-indexed
00	1	!= 1111	11	LDRSH (immediate) — post-indexed
01	0	!= 1111	10	UNALLOCATED
01	0		01	STRHT
01	0		11	UNALLOCATED
01	1		01	LDRHT
01	1		10	LDRSBT
01	1		11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) — offset

P:W	Decode fields		Instruction Details	
	o1	Rn	op2	
10	0		01	STRH (immediate) — offset
10	0		11	STRD (immediate) — offset
10	1	!= 1111	01	LDRH (immediate) — offset
10	1	!= 1111	10	LDRSB (immediate) — offset
10	1	!= 1111	11	LDRSH (immediate) — offset
11	0	!= 1111	10	LDRD (immediate) — pre-indexed
11	0		01	STRH (immediate) — pre-indexed
11	0		11	STRD (immediate) — pre-indexed
11	1	!= 1111	01	LDRH (immediate) — pre-indexed
11	1	!= 1111	10	LDRSB (immediate) — pre-indexed
11	1	!= 1111	11	LDRSH (immediate) — pre-indexed

## Multiply and Accumulate

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	0					opc	S			RdHi																		Rn
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	S	
000		MUL, MULS
001		MLA, MLAS
010	0	UMAAL
010	1	UNALLOCATED
011	0	MLS
011	1	UNALLOCATED
100		UMULL, UMULLS
101		UMLAL, UMLALS
110		SMULL, SMULLS
111		SMLAL, SMLALS

## Synchronization primitives and Load-Acquire/Store-Release

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111					0001			op0													11				1001						

Decode fields		Instruction details
op0		
0		UNALLOCATED
1		<a href="#">Load/Store Exclusive and Load-Acquire/Store-Release</a>

## Load/Store Exclusive and Load-Acquire/Store-Release

These instructions are under [Synchronization primitives and Load-Acquire/Store-Release](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	size	L	Rn				xRd				(1)	(1)	ex	ord	1	0	0	1	xRt				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
size	L	ex	ord	
00	0	0	0	STL
00	0	0	1	UNALLOCATED
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	0	1	UNALLOCATED
00	1	1	0	LDAEX
00	1	1	1	LDREX
01	0	0		UNALLOCATED
01	0	1	0	STLEXD
01	0	1	1	STREXD
01	1	0		UNALLOCATED
01	1	1	0	LDAEXD
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	0	1	UNALLOCATED
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	0	1	UNALLOCATED
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	0	1	UNALLOCATED
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	0	1	UNALLOCATED
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

## Miscellaneous

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				00010				op0		0														0		op1					

Decode fields		Instruction details
op0	op1	
00	001	UNALLOCATED
00	010	UNALLOCATED
00	011	UNALLOCATED
00	110	UNALLOCATED

01	001	BX
01	010	BXJ
01	011	BLX (register)
01	110	UNALLOCATED
10	001	UNALLOCATED
10	010	UNALLOCATED
10	011	UNALLOCATED
10	110	UNALLOCATED
11	001	CLZ
11	010	UNALLOCATED
11	011	UNALLOCATED
11	110	ERET
	111	<a href="#">Exception Generation</a>
	000	<a href="#">Move special register (register)</a>
	100	<a href="#">Cyclic Redundancy Check</a>
	101	<a href="#">Integer Saturating Arithmetic</a>

## Exception Generation

These instructions are under [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc		0	imm12												0	1	1	1	imm4			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	<a href="#">HLT</a>
01	BKPT
10	<a href="#">HVC</a>
11	<a href="#">SMC</a>

## Move special register (register)

These instructions are under [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0 0 0 1 0				opc		0		mask				Rd				(0) (0)		B	m	0	0	0	0	Rn							
cond																																			

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	B	Instruction Details
x0	0	MRS
x0	1	MRS (Banked register)
x1	0	MSR (register)
x1	1	MSR (Banked register)

## Cyclic Redundancy Check

These instructions are under [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	0	sz	0		Rn		Rd	(0)	(0)	C	(0)	0	1	0	0		Rm										
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
sz	C	
00	0	CRC32 — CRC32B
00	1	CRC32C — CRC32CB
01	0	CRC32 — CRC32H
01	1	CRC32C — CRC32CH
10	0	CRC32 — CRC32W
10	1	CRC32C — CRC32CW
11		CONSTRAINED UNPREDICTABLE

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

## Integer Saturating Arithmetic

These instructions are under [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	0	opc	0		Rn		Rd	(0)	(0)	(0)	(0)	0	1	0	1		Rm										
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc		
00		QADD
01		QSUB
10		QDADD
11		QDSUB

## Halfword Multiply and Accumulate

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	0	opc	0		Rd		Ra		Rm	1	M	N	0		Rn												
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
opc	M	N	
00			SMLABB, SMLABT, SMLATB, SMLATT
01	0	0	SMLAWB, SMLAWT — SMLAWB
01	0	1	SMULWB, SMULWT — SMULWB

Decode fields			Instruction Details
opc	M	N	
01	1	0	SMLAWB, SMLAWT — SMLAWT
01	1	1	SMULWB, SMULWT — SMULWT
10			SMLALBB, SMLALBT, SMLALTB, SMLALTT
11			SMULBB, SMULBT, SMULTB, SMULTT

## Data-processing register (immediate shift)

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				000				op0			op1																0				

The following constraints also apply to this encoding: op0:op1 != 100

Decode fields		Instruction details
op0	op1	
0x		<a href="#">Integer Data Processing (three register, immediate shift)</a>
10	1	<a href="#">Integer Test and Compare (two register, immediate shift)</a>
11		<a href="#">Logical Arithmetic (three register, immediate shift)</a>

## Integer Data Processing (three register, immediate shift)

These instructions are under [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 0 0				opc			S	Rn				Rd				imm5				stype		0	Rm				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
opc	S	Rn	imm5:stype	
000			!= 0000011	AND, ANDS (register) — shift or rotate by value
000			0000011	AND, ANDS (register) — rotate right with extend
001			!= 0000011	EOR, EORS (register) — shift or rotate by value
001			0000011	EOR, EORS (register) — rotate right with extend
010	0	!= 1101	!= 0000011	SUB, SUBS (register) — SUB, shift or rotate by value
010	0	!= 1101	0000011	SUB, SUBS (register) — SUB, rotate right with extend
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) — SUB, shift or rotate by value
010	0	1101	0000011	SUB, SUBS (SP minus register) — SUB, rotate right with extend
010	1	!= 1101	!= 0000011	SUB, SUBS (register) — SUBS, shift or rotate by value
010	1	!= 1101	0000011	SUB, SUBS (register) — SUBS, rotate right with extend
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
010	1	1101	0000011	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
011			!= 0000011	RSB, RSBS (register) — shift or rotate by value
011			0000011	RSB, RSBS (register) — rotate right with extend
100	0	!= 1101	!= 0000011	ADD, ADDS (register) — ADD, shift or rotate by value
100	0	!= 1101	0000011	ADD, ADDS (register) — ADD, rotate right with extend
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) — ADD, shift or rotate by value
100	0	1101	0000011	ADD, ADDS (SP plus register) — ADD, rotate right with extend

Decode fields				Instruction Details
opc	S	Rn	imm5:type	
100	1	!= 1101	!= 0000011	ADD, ADDS (register) — ADDS, shift or rotate by value
100	1	!= 1101	0000011	ADD, ADDS (register) — ADDS, rotate right with extend
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
100	1	1101	0000011	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
101			!= 0000011	ADC, ADCS (register) — shift or rotate by value
101			0000011	ADC, ADCS (register) — rotate right with extend
110			!= 0000011	SBC, SBCS (register) — shift or rotate by value
110			0000011	SBC, SBCS (register) — rotate right with extend
111			!= 0000011	RSC, RSCS (register) — shift or rotate by value
111			0000011	RSC, RSCS (register) — rotate right with extend

### Integer Test and Compare (two register, immediate shift)

These instructions are under [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	0	opc	1		Rn	(0)	(0)	(0)	(0)		imm5		stype	0		Rm											
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	imm5:type	
00	!= 0000011	TST (register) — shift or rotate by value
00	0000011	TST (register) — rotate right with extend
01	!= 0000011	TEQ (register) — shift or rotate by value
01	0000011	TEQ (register) — rotate right with extend
10	!= 0000011	CMP (register) — shift or rotate by value
10	0000011	CMP (register) — rotate right with extend
11	!= 0000011	CMN (register) — shift or rotate by value
11	0000011	CMN (register) — rotate right with extend

### Logical Arithmetic (three register, immediate shift)

These instructions are under [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	1	opc	S		Rn		Rd				imm5		stype	0		Rm											
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	imm5:type	
00	!= 0000011	ORR, ORRS (register) — shift or rotate by value
00	0000011	ORR, ORRS (register) — rotate right with extend
01	!= 0000011	MOV, MOVS (register) — shift or rotate by value
01	0000011	MOV, MOVS (register) — rotate right with extend
10	!= 0000011	BIC, BICS (register) — shift or rotate by value
10	0000011	BIC, BICS (register) — rotate right with extend
11	!= 0000011	MVN, MVNS (register) — shift or rotate by value

Decode fields		Instruction Details
opc	imm5:type	
11	0000011	MVN, MVNS (register) — rotate right with extend

## Data-processing register (register shift)

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				000			op0				op1												0				1				

The following constraints also apply to this encoding: op0:op1 != 100

Decode fields		Instruction details
op0	op1	
0x		<a href="#">Integer Data Processing (three register, register shift)</a>
10	1	<a href="#">Integer Test and Compare (two register, register shift)</a>
11		<a href="#">Logical Arithmetic (three register, register shift)</a>

## Integer Data Processing (three register, register shift)

These instructions are under [Data-processing register \(register shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 0 0				opc			S	Rn				Rd				Rs				0	stype		1	Rm			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc		
000		AND, ANDS (register-shifted register)
001		EOR, EORS (register-shifted register)
010		SUB, SUBS (register-shifted register)
011		RSB, RSBS (register-shifted register)
100		ADD, ADDS (register-shifted register)
101		ADC, ADCS (register-shifted register)
110		SBC, SBCS (register-shifted register)
111		RSC, RSCS (register-shifted register)

## Integer Test and Compare (two register, register shift)

These instructions are under [Data-processing register \(register shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				0 0 0 1 0				opc		1		Rn				(0)	(0)	(0)	(0)	Rs				0		stype		1		Rm			
cond																																	

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc		
00		TST (register-shifted register)
01		TEQ (register-shifted register)

Decode fields	Instruction Details
opc	
10	CMP (register-shifted register)
11	CMN (register-shifted register)

### Logical Arithmetic (three register, register shift)

These instructions are under [Data-processing register \(register shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	opc		S	Rn				Rd				Rs				0	stype		1	Rm			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
opc	
00	ORR, ORRS (register-shifted register)
01	MOV, MOVS (register-shifted register)
10	BIC, BICS (register-shifted register)
11	MVN, MVNS (register-shifted register)

### Data-processing immediate

These instructions are under [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111			001			op0				op1																					

Decode fields	Instruction details	
op0	op1	
0x		<a href="#">Integer Data Processing (two register and immediate)</a>
10	00	<a href="#">Move Halfword (immediate)</a>
10	10	<a href="#">Move Special Register and Hints (immediate)</a>
10	x1	<a href="#">Integer Test and Compare (one register and immediate)</a>
11		<a href="#">Logical Arithmetic (two register and immediate)</a>

### Integer Data Processing (two register and immediate)

These instructions are under [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	1	0		opc	S				Rn				Rd																
cond																															
																imm12															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
opc	S	Rn	
000			AND, ANDS (immediate)
001			EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) — SUB
010	0	1101	SUB, SUBS (SP minus immediate) — SUB
010	0	1111	ADR — A2

Decode fields			Instruction Details
opc	S	Rn	
010	1	!= 1101	SUB, SUBS (immediate) — SUBS
010	1	1101	SUB, SUBS (SP minus immediate) — SUBS
011			RSB, RSBS (immediate)
100	0	!= 11x1	ADD, ADDS (immediate) — ADD
100	0	1101	ADD, ADDS (SP plus immediate) — ADD
100	0	1111	ADR — A1
100	1	!= 1101	ADD, ADDS (immediate) — ADDS
100	1	1101	ADD, ADDS (SP plus immediate) — ADDS
101			ADC, ADCS (immediate)
110			SBC, SBCS (immediate)
111			RSC, RSCS (immediate)

### Move Halfword (immediate)

These instructions are under [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	1	1	0	H	0	0				imm4																			
cond																Rd				imm12											

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
H		
0		MOV, MOVS (immediate)
1		MOVT

### Move Special Register and Hints (immediate)

These instructions are under [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	1	1	0	R	1	0				imm4	(1)	(1)	(1)	(1)															
cond																				imm12											

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details	Feature
R:imm4	imm12		
!= 00000		MSR (immediate)	-
00000	xxxx00000000	NOP	-
00000	xxxx00000001	YIELD	-
00000	xxxx00000010	WFE	-
00000	xxxx00000011	WFI	-
00000	xxxx00000100	SEV	-
00000	xxxx00000101	SEVL	-
00000	xxxx0000011x	Reserved hint, behaves as NOP	-
00000	xxxx00001xxx	Reserved hint, behaves as NOP	-
00000	xxxx00010000	ESB	FEAT_RAS
00000	xxxx00010001	Reserved hint, behaves as NOP	-
00000	xxxx00010010	<a href="#">TSB CSYNC</a>	FEAT_TRF

Decode fields		Instruction Details	Feature
R:imm4	imm12		
00000	XXXX00010011	Reserved hint, behaves as NOP	-
00000	XXXX00010100	CSDB	-
00000	XXXX00010101	Reserved hint, behaves as NOP	-
00000	XXXX0001011x	Reserved hint, behaves as NOP	-
00000	XXXX00011xxx	Reserved hint, behaves as NOP	-
00000	XXXX001xxxxx	Reserved hint, behaves as NOP	-
00000	XXXX01xxxxxx	Reserved hint, behaves as NOP	-
00000	XXXX10xxxxxx	Reserved hint, behaves as NOP	-
00000	XXXX110xxxxx	Reserved hint, behaves as NOP	-
00000	XXXX1110xxxx	Reserved hint, behaves as NOP	-
00000	XXXX1111xxxx	DBG	-

### Integer Test and Compare (one register and immediate)

These instructions are under [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 1 1 0				opc		1		Rn				0	0	0	0	imm12											
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
opc	
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

### Logical Arithmetic (two register and immediate)

These instructions are under [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 1 1 1			opc		S	Rn					Rd				imm12												
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
opc	
00	ORR, ORRS (immediate)
01	MOV, MOVS (immediate)
10	BIC, BICS (immediate)
11	MVN, MVNS (immediate)

### Load/Store Word, Unsigned Byte (immediate, literal)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	1	0	P	U	o2	W	o1	Rn				Rt				imm12															
cond																																			

The following constraints also apply to this encoding: `cond != 1111 && cond != 1111`

Decode fields				Instruction Details
P:W	o2	o1	Rn	
!= 01	0	1	1111	LDR (literal)
!= 01	1	1	1111	LDRB (literal)
00	0	0		STR (immediate) — post-indexed
00	0	1	!= 1111	LDR (immediate) — post-indexed
00	1	0		STRB (immediate) — post-indexed
00	1	1	!= 1111	LDRB (immediate) — post-indexed
01	0	0		STRT
01	0	1		LDRT
01	1	0		STRBT
01	1	1		LDRBT
10	0	0		STR (immediate) — offset
10	0	1	!= 1111	LDR (immediate) — offset
10	1	0		STRB (immediate) — offset
10	1	1	!= 1111	LDRB (immediate) — offset
11	0	0		STR (immediate) — pre-indexed
11	0	1	!= 1111	LDR (immediate) — pre-indexed
11	1	0		STRB (immediate) — pre-indexed
11	1	1	!= 1111	LDRB (immediate) — pre-indexed

### Load/Store Word, Unsigned Byte (register)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	P	U	o2	W	o1	Rn				Rt				imm5				stype		0	Rm				
cond																															

The following constraints also apply to this encoding: `cond != 1111 && cond != 1111`

Decode fields				Instruction Details
P	o2	W	o1	
0	0	0	0	STR (register) — post-indexed
0	0	0	1	LDR (register) — post-indexed
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) — post-indexed
0	1	0	1	LDRB (register) — post-indexed
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0		0	STR (register) — pre-indexed
1	0		1	LDR (register) — pre-indexed
1	1		0	STRB (register) — pre-indexed
1	1		1	LDRB (register) — pre-indexed

### Media instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				011			op0																op1			1					

Decode fields		Instruction details
op0	op1	
00xxx		<a href="#">Parallel Arithmetic</a>
01000	101	SEL
01000	001	UNALLOCATED
01000	xx0	PKHBT, PKHTB
01001	x01	UNALLOCATED
01001	xx0	UNALLOCATED
0110x	x01	UNALLOCATED
0110x	xx0	UNALLOCATED
01x10	001	<a href="#">Saturate 16-bit</a>
01x10	101	UNALLOCATED
01x11	x01	<a href="#">Reverse Bit/Byte</a>
01x1x	xx0	<a href="#">Saturate 32-bit</a>
01xxx	111	UNALLOCATED
01xxx	011	<a href="#">Extend and Add</a>
10xxx		<a href="#">Signed multiply, Divide</a>
11000	000	<a href="#">Unsigned Sum of Absolute Differences</a>
11000	100	UNALLOCATED
11001	x00	UNALLOCATED
1101x	x00	UNALLOCATED
110xx	111	UNALLOCATED
1110x	111	UNALLOCATED
1110x	x00	<a href="#">Bitfield Insert</a>
11110	111	UNALLOCATED
11111	111	<a href="#">Permanently UNDEFINED</a>
1111x	x00	UNALLOCATED
11x0x	x10	UNALLOCATED
11x1x	x10	<a href="#">Bitfield Extract</a>
11xxx	011	UNALLOCATED
11xxx	x01	UNALLOCATED

## Parallel Arithmetic

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0 1 1 0 0				op1				Rn				Rd				(1)	(1)	(1)	(1)	B	op2		1		Rm			
cond																																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
op1	B	op2	
000			UNALLOCATED
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	UNALLOCATED

Decode fields			Instruction Details
op1	B	op2	
001	1	10	UNALLOCATED
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	UNALLOCATED
010	1	10	UNALLOCATED
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	UNALLOCATED
011	1	10	UNALLOCATED
011	1	11	SHSUB8
100			UNALLOCATED
101	0	00	UADD16
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	UNALLOCATED
101	1	10	UNALLOCATED
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	UNALLOCATED
110	1	10	UNALLOCATED
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	UNALLOCATED
111	1	10	UNALLOCATED
111	1	11	UHSUB8

**Saturate 16-bit**

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	U	1	0	sat_imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
U	
0	SSAT16
1	USAT16

## Reverse Bit/Byte

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	o1	1	1	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	o2	0	1	1	Rm			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
o1	o2	
0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

## Saturate 32-bit

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	U	1	sat imm				Rd				imm5				sh	0	1	Rn					
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
U	
0	SSAT
1	USAT

## Extend and Add

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	1	1	0	1	U	op			Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond																																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
U	op	Rn	
0	00	!= 1111	SXTAB16

Decode fields			Instruction Details
U	op	Rn	
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

### Signed multiply, Divide

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	op1			Rd			Ra			Rm			op2			1	Rn						
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
op1	Ra	op2	
000	!= 1111	000	SMLAD, SMLADX — SMLAD
000	!= 1111	001	SMLAD, SMLADX — SMLADX
000	!= 1111	010	SMLSD, SMLSDX — SMLSD
000	!= 1111	011	SMLSD, SMLSDX — SMLSDX
000		1xx	UNALLOCATED
000	1111	000	SMUAD, SMUADX — SMUAD
000	1111	001	SMUAD, SMUADX — SMUADX
000	1111	010	SMUSD, SMUSDX — SMUSD
000	1111	011	SMUSD, SMUSDX — SMUSDX
001		000	SDIV
001		!= 000	UNALLOCATED
010			UNALLOCATED
011		000	UDIV
011		!= 000	UNALLOCATED
100		000	SMLALD, SMLALDX — SMLALD
100		001	SMLALD, SMLALDX — SMLALDX
100		010	SMLSLD, SMLSLDX — SMLSLD
100		011	SMLSLD, SMLSLDX — SMLSLDX
100		1xx	UNALLOCATED
101	!= 1111	000	SMMLA, SMMLAR — SMMLA
101	!= 1111	001	SMMLA, SMMLAR — SMMLAR
101		01x	UNALLOCATED
101		10x	UNALLOCATED
101		110	SMMLS, SMMLSR — SMMLS
101		111	SMMLS, SMMLSR — SMMLSR

Decode fields			Instruction Details
op1	Ra	op2	
101	1111	000	SMMUL, SMMULR — SMMUL
101	1111	001	SMMUL, SMMULR — SMMULR
11x			UNALLOCATED

## Unsigned Sum of Absolute Differences

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	1	1	1	1	0	0	0					Rd																		Rn
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
Ra	
!= 1111	USADA8
1111	USAD8

## Bitfield Insert

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	1	1	1	1	1	0						msb																		Rn
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
Rn	
!= 1111	<a href="#">BFI</a>
1111	<a href="#">BFC</a>

## Permanently UNDEFINED

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	1	1	1	imm12											1	1	1	1	imm4				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
cond	
0xxx	UNALLOCATED
10xx	UNALLOCATED
110x	UNALLOCATED
1110	UDF

## Bitfield Extract

These instructions are under [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 1 1 1 1				U 1		widthm1						Rd				lsb				1 0 1			Rn				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
U	
0	<a href="#">SBFX</a>
1	<a href="#">UBFX</a>

## Branch, branch with link, and block data transfer

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				10		op0																									

Decode fields		Instruction details
cond	op0	
1111	0	<a href="#">Exception Save/Restore</a>
!= 1111	0	<a href="#">Load/Store Multiple</a>
	1	<a href="#">Branch (immediate)</a>

## Exception Save/Restore

These instructions are under [Branch, branch with link, and block data transfer](#).

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

Decode fields	Instruction Details			
P	U	S	L	
		0	0	UNALLOCATED
0	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement After
0	0	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Decrement After</a>
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment After
0	1	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Increment After</a>
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement Before
1	0	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Decrement Before</a>
		1	1	UNALLOCATED
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment Before
1	1	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Increment Before</a>

## Load/Store Multiple

These instructions are under [Branch, branch with link, and block data transfer](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1 0 0		P	U	op	W	L	Rn					register_list															
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				register_list	Instruction Details
P	U	op	L		
0	0	0	0		STMDA, STMED
0	0	0	1		LDMDA, LDMFA
0	1	0	0		STM, STMIA, STMEA
0	1	0	1		LDM, LDMIA, LDMFD
		1	0		STM (User registers)
1	0	0	0		STMDB, STMFD
1	0	0	1		LDMDB, LDMEA
		1	1	0xxxxxxxxxxxxxxxxxx	LDM (User registers)
1	1	0	0		STMIB, STMFA
1	1	0	1		LDMIB, LDMED
		1	1	1xxxxxxxxxxxxxxxxxx	LDM (exception return)

### Branch (immediate)

These instructions are under [Branch, branch with link, and block data transfer](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	H	imm24																							

Decode fields		H	Instruction Details
cond			
!= 1111	0	B	
!= 1111	1	BL, BLX (immediate) — A1	
1111		BL, BLX (immediate) — A2	

### System register access, Advanced SIMD, floating-point, and Supervisor call

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				11		op0														op1						op2					

Decode fields				Instruction details
cond	op0	op1	op2	
	0x	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			<a href="#">Supervisor call</a>
1111	!= 11	1x		<a href="#">Unconditional Advanced SIMD and floating-point instructions</a>
!= 1111	0x	1x		<a href="#">Advanced SIMD and System register load/store and 64-bit move</a>
!= 1111	10	1x	1	<a href="#">Advanced SIMD and System register 32-bit move</a>
!= 1111	10	10	0	<a href="#">Floating-point data-processing</a>
!= 1111	10	11	0	UNALLOCATED

### Supervisor call

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1111																											

**Decode fields**      **Instruction details**

cond	
1111	UNALLOCATED
!= 1111	SVC

## Unconditional Advanced SIMD and floating-point instructions

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111111						op0			op1								1	op2	op3		op4		op5								

The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			<a href="#">Advanced SIMD three registers of the same length extension</a>
100		0	!= 00	0	0	<a href="#">Floating-point conditional select</a>
101	00xxxx	0	!= 00		0	<a href="#">Floating-point minNum/maxNum</a>
101	110000	0	!= 00	1	0	<a href="#">Floating-point extraction and insertion</a>
101	111xxx	0	!= 00	1	0	<a href="#">Floating-point directed convert to integer</a>
10x		0	00			<a href="#">Advanced SIMD and floating-point multiply with accumulate</a>
10x		1	0x			<a href="#">Advanced SIMD and floating-point dot product</a>

## Advanced SIMD three registers of the same length extension

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	op1	D	op2	Vn			Vd			1	op3	0	op4	N	Q	M	U	Vm							

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAB, VFMA (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	1			UNALLOCATED	-

### Floating-point conditional select

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc			Vn				Vd			1	0	!= 00	N	0	M	0			Vm		
																size															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	Instruction Details	Feature
01	VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11	VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

## Floating-point minNum/maxNum

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn			Vd			1			0	!= 00	N	op	M	0	Vm				
size																															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

## Floating-point extraction and insertion

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd			1			0	!= 00	op	1	M	0	Vm			
																size															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

## Floating-point directed convert to integer

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	o1	RM	Vd			1			0	!= 00	op	1	M	0	Vm				
																size															

The following constraints also apply to this encoding: size != 00 && size != 00

o1	Decode fields RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-

Decode fields				Instruction Details	Feature
o1	RM	size	op		
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-
1	11	11		VCVTM (floating-point) — double-precision scalar	-

### Advanced SIMD and floating-point multiply with accumulate

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2					Vn				Vd			1	0	0	0	N	Q	M	U		Vm

Decode fields				Instruction Details	Feature
op1	op2	Q	U		
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00		1	VFMAL (by scalar)	FEAT_FHM
0	01		1	VFMSL (by scalar)	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAB, VFMA (BFloat16, by scalar)	FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

### Advanced SIMD and floating-point dot product

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2					Vn				Vd			1	1	0	op4	N	Q	M	U		Vm

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	00	0			UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			UNALLOCATED	-

## Advanced SIMD and System register load/store and 64-bit move

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1111					110															1		op1								

Decode fields		Instruction details
op0	op1	
00x0	0x	<a href="#">Advanced SIMD and floating-point 64-bit move</a>
00x0	11	<a href="#">System register 64-bit move</a>
!= 00x0	0x	<a href="#">Advanced SIMD and floating-point load/store</a>
!= 00x0	11	<a href="#">System register load/store</a>
	10	UNALLOCATED

## Advanced SIMD and floating-point 64-bit move

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1 1 0 0 0				D 0		op		Rt2				Rt				1 0		size		opc2		M o3		Vm			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields					Instruction Details
D	op	size	opc2	o3	
0					UNALLOCATED
1				0	UNALLOCATED
1		0x	00	1	UNALLOCATED
1			01		UNALLOCATED
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers

Decode fields					Instruction Details
D	op	size	opc2	o3	
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers
1			1x		UNALLOCATED
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers

## System register 64-bit move

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	D	0	L	Rt2				Rt				1	1	1	cp15	opc1			CRm				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
D	L	
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

## Advanced SIMD and floating-point load/store

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	0	P	U	D	W	L							Rn					Vd		1	0	size						imm8	
cond																															

The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

Decode fields						imm8	Instruction Details	Feature
P	U	W	L	Rn	size			
0	0	1					UNALLOCATED	-
0	1				0x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
0	1		0		11	XXXXXXXX0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
0	1		0		11	XXXXXXXX1	FSTMDBX, FSTMIA — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
0	1		1		11	XXXXXXXX0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
0	1		1		11	XXXXXXXX1	FLDM*X (FLDMDBX, FLDMIA) — Increment After	-
1		0	0		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-

Decode fields						Instruction Details		Feature
P	U	W	L	Rn	size	imm8		
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
1	0	1	0		11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
1	0	1	0		11	xxxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
1	0	1	1		11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
1	0	1	1		11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1		0	1	1111	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

## System register load/store

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	0	P	U	D	W	L			Rn				CRd				1	1	1	cp15									imm8
cond																															

The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

Decode fields						Instruction Details	
P:U:W	D	L	Rn	CRd	cp15		
!= 000	0			!= 0101	0	UNALLOCATED	
!= 000	0	1	1111	0101	0	LDC (literal)	
!= 000					1	UNALLOCATED	
!= 000	1			0101	0	UNALLOCATED	
0x1	0	0		0101	0	STC — post-indexed	
0x1	0	1	!= 1111	0101	0	LDC (immediate) — post-indexed	
010	0	0		0101	0	STC — unindexed	
010	0	1	!= 1111	0101	0	LDC (immediate) — unindexed	
1x0	0	0		0101	0	STC — offset	
1x0	0	1	!= 1111	0101	0	LDC (immediate) — offset	
1x1	0	0		0101	0	STC — pre-indexed	
1x1	0	1	!= 1111	0101	0	LDC (immediate) — pre-indexed	

## Advanced SIMD and System register 32-bit move

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1110				op0												1		op1				1					

Decode fields		Instruction details	Architecture version
op0	op1		
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	<a href="#">Floating-point move special register</a>	-
	011	<a href="#">Advanced SIMD 8/16/32-bit element move/duplicate</a>	-
	10x	UNALLOCATED	-
	11x	<a href="#">System register 32-bit move</a>	-

### Floating-point move special register

These instructions are under [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
!= 1111				1	1	1	0	1	1	1	L	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)						
cond																																					

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
L		
0		VMSR
1		VMRS

### Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
!= 1111				1 1 1 0		opc1			L	Vn				Rt				1 0 1 1		N	opc2		1	(0)	(0)	(0)	(0)										
cond																																					

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
opc1	L	opc2	
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

### System register 32-bit move

These instructions are under [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				1	1	1	0	opc1			L	CRn				Rt				1	1	1	cp15	opc2		1	CRm					
cond																																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
L	
0	MCR
1	MRC

## Floating-point data-processing

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1110					op0											10				op1			0				

Decode fields	op0	op1	Instruction details
1x11	1		<a href="#">Floating-point data-processing (two registers)</a>
1x11	0		<a href="#">Floating-point move immediate</a>
!= 1x11			<a href="#">Floating-point data-processing (three registers)</a>

## Floating-point data-processing (two registers)

These instructions are under [Floating-point data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	o1	opc2			Vd			1 0		size	o3	1	M	0	Vm					
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

o1	Decode fields	opc2	size	o3	Instruction Details	Feature
		00			UNALLOCATED	-
0	000	01	0		UNALLOCATED	-
0	000	01	1		VABS — half-precision scalar	FEAT_FP16
0	000	10	0		VMOV (register) — single-precision scalar	-
0	000	10	1		VABS — single-precision scalar	-
0	000	11	0		VMOV (register) — double-precision scalar	-
0	000	11	1		VABS — double-precision scalar	-
0	001	01	0		VNEG — half-precision scalar	FEAT_FP16
0	001	01	1		VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0		VNEG — single-precision scalar	-
0	001	10	1		VSQRT — single-precision scalar	-
0	001	11	0		VNEG — double-precision scalar	-
0	001	11	1		VSQRT — double-precision scalar	-
0	010	01			UNALLOCATED	-
0	010	10	0		VCVTB — half-precision to single-precision	-
0	010	10	1		VCVTT — half-precision to single-precision	-
0	010	11	0		VCVTB — half-precision to double-precision	-
0	010	11	1		VCVTT — half-precision to double-precision	-
0	011	01	0		VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1		VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0		VCVTB — single-precision to half-precision	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

### Floating-point move immediate

These instructions are under [Floating-point data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	imm4H				Vd				1 0		size	(0)	0	(0)	0	imm4L				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details	Feature
size			
00		UNALLOCATED	-
01		VMOV (immediate) — half-precision scalar	FEAT_FP16
10		VMOV (immediate) — single-precision scalar	-
11		VMOV (immediate) — double-precision scalar	-

### Floating-point data-processing (three registers)

These instructions are under [Floating-point data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				1	1	1	0	o0	D	o1	Vn					Vd					1	0	size	N	o2	M	0	Vm				
cond																																

The following constraints also apply to this encoding: cond != 1111 && o0:D:o1 != 1x11 && cond != 1111

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
!= 111	00		UNALLOCATED	-
000	01	0	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

## Unconditional instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11110					op0					op1																					

Decode fields		Instruction details
op0	op1	
00x		<a href="#">Miscellaneous</a>
01x		<a href="#">Advanced SIMD data-processing</a>
1xx	1	<a href="#">Memory hints and barriers</a>
100	0	<a href="#">Advanced SIMD element or structure load/store</a>
101	0	UNALLOCATED
11x	0	UNALLOCATED

## Miscellaneous

These instructions are under [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111000								op0								op1															

Decode fields		Instruction details	Architecture version
op0	op1		
0xxxx		UNALLOCATED	-
10000	xx0x	<a href="#">Change Process State</a>	-
10001	1000	UNALLOCATED	-
10001	x100	UNALLOCATED	-
10001	xx01	UNALLOCATED	-
10001	0000	SETPAN	FEAT_PAN
1000x	0111	UNALLOCATED	-
10010	0111	CONSTRAINED UNPREDICTABLE	-
10011	0111	UNALLOCATED	-
1001x	xx0x	UNALLOCATED	-
100xx	0011	UNALLOCATED	-
100xx	0x10	UNALLOCATED	-
100xx	1x1x	UNALLOCATED	-
101xx		UNALLOCATED	-
11xxx		UNALLOCATED	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

## Change Process State

These instructions are under [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	op	(0)	(0)	(0)	(0)	(0)	(0)	(0)	E	A	I	F	0	mode				

Decode fields						Instruction Details
imod	M	op	I	F	mode	
		1	0	0	0xxxx	SETEND
00	1	0				CPS, CPSID, CPSIE — change mode
10		0				CPS, CPSID, CPSIE — interrupt enable and change mode
		1	0	0	1xxxx	UNALLOCATED
		1	0	1		UNALLOCATED
		1	1			UNALLOCATED
11		0				CPS, CPSID, CPSIE — interrupt disable and change mode

## Advanced SIMD data-processing

These instructions are under [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1111001								op0																	op1								

Decode fields		Instruction details
op0	op1	
0		<a href="#">Advanced SIMD three registers of the same length</a>
1	0	<a href="#">Advanced SIMD two registers, or three registers of different lengths</a>
1	1	<a href="#">Advanced SIMD shifts and immediate generation</a>

**Advanced SIMD three registers of the same length**

These instructions are under [Advanced SIMD data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn			Vd			opc			N		Q	M	o1	Vm						

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) — A2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	-
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) — A1	-
		0011		1	VCGE (register) — A1	-
0	01	1100		0	SHA1P	-
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	<a href="#">VRSHL</a>	-
		0101		1	<a href="#">VQRSHL</a>	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	-
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-

U	Decode fields			o1	Instruction Details	Feature
	size	opc	Q			
1	0x	1110		0	VCGE (register) — A2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	-
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	-
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) — A2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) — A1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

## Advanced SIMD two registers, or three registers of different lengths

These instructions are under [Advanced SIMD data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
							1111001		op0	1			op1								op2				op3			0			

op0	Decode fields			op3	Instruction details
	op1	op2			
0	11				VEXT (byte elements)
1	11	0x			<a href="#">Advanced SIMD two registers misc</a>
1	11	10			VTBL, VTBX
1	11	11			<a href="#">Advanced SIMD duplicate (scalar)</a>
	!= 11			0	<a href="#">Advanced SIMD three registers of different lengths</a>
	!= 11			1	<a href="#">Advanced SIMD two registers and a scalar</a>

## Advanced SIMD two registers misc

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	opc1	Vd			0	opc2			Q	M	0	Vm							

size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
	00	0000		VREV64	-
	00	0001		VREV32	-
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	-
	00	0110	1	AESD	-
	00	0111	0	AESMC	-
	00	0111	1	AESIMC	-
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VC GT (immediate #0)	-
	01	x001		VC GE (immediate #0)	-
	01	x010		VC EQ (immediate #0)	-
	01	x011		VC LE (immediate #0)	-
	01	x100		VC LT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	-
	10	0111	1	SHA256SU0	-
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-

Decode fields				Instruction Details	Feature
size	opc1	opc2	Q		
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRTE	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

### Advanced SIMD duplicate (scalar)

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	imm4			Vd			1 1			opc		Q	M	0	Vm					

Decode fields opc	Instruction Details
000	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

### Advanced SIMD three registers of different lengths

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 11	Vn			Vd			opc			N	0	M	0	Vm							

size

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields U	opc	Instruction Details
	0000	VADDL
	0001	VADDW
	0010	VSUBL
0	0100	VADDHN
	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)

Decode fields	Instruction Details
U      opc	
1      1001	UNALLOCATED
1      1011	UNALLOCATED
1      1101	UNALLOCATED
1111	UNALLOCATED

### Advanced SIMD two registers and a scalar

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn				Vd				opc				N	1	M	0	Vm				
size																															

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields	Instruction Details	Feature
Q      opc		
000x	VMLA (by scalar)	-
0      0011	VQDMLAL	-
0010	VMLAL (by scalar)	-
0      0111	VQDMLSL	-
010x	VMLS (by scalar)	-
0      1011	VQDMULL	-
0110	VMLSL (by scalar)	-
100x	VMUL (by scalar)	-
1      0011	UNALLOCATED	-
1010	VMULL (by scalar)	-
1      0111	UNALLOCATED	-
1100	VQDMULH	-
1101	VQRDMULH	-
1      1011	UNALLOCATED	-
1110	VQRDMLAH	FEAT_RDM
1111	VQRDMLSH	FEAT_RDM

### Advanced SIMD shifts and immediate generation

These instructions are under [Advanced SIMD data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111001									1		op0															1					

Decode fields	Instruction details
op0	
000xxxxxxxxxxxx0	<a href="#">Advanced SIMD one register and modified immediate</a>
!= 000xxxxxxxxxxxx0	<a href="#">Advanced SIMD two registers and shift amount</a>

### Advanced SIMD one register and modified immediate

These instructions are under [Advanced SIMD shifts and immediate generation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			cmode			0	Q	op	1	imm4					

Decode fields		Instruction Details
cmode	op	
0xx0	0	VMOV (immediate) — A1
0xx0	1	VMVN (immediate) — A1
0xx1	0	VORR (immediate) — A1
0xx1	1	VBIC (immediate) — A1
10x0	0	VMOV (immediate) — A3
10x0	1	VMVN (immediate) — A2
10x1	0	VORR (immediate) — A2
10x1	1	VBIC (immediate) — A2
11xx	0	VMOV (immediate) — A4
110x	1	VMVN (immediate) — A3
1110	1	VMOV (immediate) — A5
1111	1	UNALLOCATED

### Advanced SIMD two registers and shift amount

These instructions are under [Advanced SIMD shifts and immediate generation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm3H			imm3L			Vd			opc			L	Q	M	1	Vm					

The following constraints also apply to this encoding: imm3H:imm3L:Vd:opc:L != 000xxxxxxxxxxx0

Decode fields					Instruction Details
U	imm3H:L	imm3L	opc	Q	
	!= 0000		0000		VSHR
	!= 0000		0001		VSRA
	!= 0000	000	1010	0	VMOVL
	!= 0000		0010		VRSRHR
	!= 0000		0011		VRSRA
	!= 0000		0111		VQSHL, VQSHLU (immediate) — VQSHL
	!= 0000		1001	0	VQSHRN, VQSHRUN — VQSHRN
	!= 0000		1001	1	VQRSHRN, VQRSHRUN — VQRSHRN
	!= 0000		1010	0	VSHLL
	!= 0000		11xx		VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000		0101		VSHL (immediate)
0	!= 0000		1000	0	VSHRN
0	!= 0000		1000	1	VRSHRN
1	!= 0000		0100		VSRI
1	!= 0000		0101		VSLI
1	!= 0000		0110		VQSHL, VQSHLU (immediate) — VQSHLU
1	!= 0000		1000	0	VQSHRN, VQSHRUN — VQSHRUN
1	!= 0000		1000	1	VQRSHRN, VQRSHRUN — VQRSHRUN

### Memory hints and barriers

These instructions are under [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111101						op0						1															op1				

Decode fields		Instruction details
op0	op1	
00xx1		CONSTRAINED UNPREDICTABLE
01001		CONSTRAINED UNPREDICTABLE
01011		<a href="#">Barriers</a>
011x1		CONSTRAINED UNPREDICTABLE
0xxx0		<a href="#">Preload (immediate)</a>
1xxx0	0	<a href="#">Preload (register)</a>
1xxx1	0	CONSTRAINED UNPREDICTABLE
1xxxx	1	UNALLOCATED

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

## Barriers

These instructions are under [Memory hints and barriers](#).

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

Decode fields		Instruction Details	Feature
opcode	option		
0000		CONSTRAINED UNPREDICTABLE	-
0001		CLREX	-
001x		CONSTRAINED UNPREDICTABLE	-
0100	!= 0x00	<a href="#">DSB</a>	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101		DMB	-
0110		ISB	-
0111		<a href="#">SB</a>	FEAT_SB
1xxx		CONSTRAINED UNPREDICTABLE	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

## Preload (immediate)

These instructions are under [Memory hints and barriers](#).

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

Decode fields			Instruction Details
D	R	Rn	
0	0		Reserved hint, behaves as NOP
0	1		PLI (immediate, literal)
1		1111	PLD (literal)
1	0	!= 1111	PLD, PLDW (immediate) — preload write
1	1	!= 1111	PLD, PLDW (immediate) — preload read

**Preload (register)**

These instructions are under [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	D	U	o2	0	1			Rn		(1)	(1)	(1)	(1)			imm5		stype	0			Rm			

Decode fields			Instruction Details
D	o2	imm5:stype	
0	0		Reserved hint, behaves as NOP
0	1	!= 0000011	PLI (register) — shift or rotate by value
0	1	0000011	PLI (register) — rotate right with extend
1	0	!= 0000011	PLD, PLDW (register) — preload write, optional shift or rotate
1	0	0000011	PLD, PLDW (register) — preload write, rotate right with extend
1	1	!= 0000011	PLD, PLDW (register) — preload read, optional shift or rotate
1	1	0000011	PLD, PLDW (register) — preload read, rotate right with extend

**Advanced SIMD element or structure load/store**

These instructions are under [Unconditional instructions](#).

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

Decode fields		Instruction details
op0	op1	
0		<a href="#">Advanced SIMD load/store multiple structures</a>
1	11	<a href="#">Advanced SIMD load single structure to all lanes</a>
1	!= 11	<a href="#">Advanced SIMD load/store single structure to one lane</a>

**Advanced SIMD load/store multiple structures**

These instructions are under [Advanced SIMD element or structure load/store](#).

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

Decode fields			Instruction Details
L	itype	Rm	
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)

Decode fields			Instruction Details
L	itype	Rm	
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

### Advanced SIMD load single structure to all lanes

These instructions are under [Advanced SIMD element or structure load/store](#).

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

Decode fields			Instruction Details
L	N	a	
0			UNALLOCATED
1	00		!= 11x1
1	00		1101
1	00		1111

Decode fields				Instruction Details
L	N	a	Rm	
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

### Advanced SIMD load/store single structure to one lane

These instructions are under [Advanced SIMD element or structure load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	L	0	Rn			Vd			!= 11			N	index_align			Rm			size			

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)

Decode fields				Instruction Details
L	size	N	Rm	
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

## Top-level encodings for A32

Internal version only: isa **v01\_27**~~v01\_26~~, pseudocode **v2022-03\_rel**~~v2021-12\_to\_suppress\_diffs in 2022-03\_RC1~~; Build timestamp: **2022-03-29T10**~~2022-03-08T10:4611~~

Copyright © **2010-2022**~~2010-2021~~ Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**(old)**

**htmldiff from-**

**(new)**

(old)

htmldiff from-

(new)

## Top-level encodings for T32

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0					op1																										

Decode fields		Instruction details
op0	op1	
!= 111		<a href="#">16-bit</a>
111	00	B — T2
111	!= 00	<a href="#">32-bit</a>

### 16-bit

These instructions are under the [top-level](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0															

The following constraints also apply to this encoding: op0<5:3> != 111

Decode fields	Instruction details
op0	
00XXXX	<a href="#">Shift (immediate), add, subtract, move, and compare</a>
010000	<a href="#">Data-processing (two low registers)</a>
010001	<a href="#">Special data instructions and branch and exchange</a>
01001x	LDR (literal) — T1
0101xx	<a href="#">Load/store (register offset)</a>
011xxx	<a href="#">Load/store word/byte (immediate offset)</a>
1000xx	<a href="#">Load/store halfword (immediate offset)</a>
1001xx	<a href="#">Load/store (SP-relative)</a>
1010xx	<a href="#">Add PC/SP (immediate)</a>
1011xx	<a href="#">Miscellaneous 16-bit instructions</a>
1100xx	<a href="#">Load/store multiple</a>
1101xx	<a href="#">Conditional branch, and Supervisor Call</a>

### Shift (immediate), add, subtract, move, and compare

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00	op0	op1	op2												

Decode fields			Instruction details
op0	op1	op2	
0	11	0	<a href="#">Add, subtract (three low registers)</a>
0	11	1	<a href="#">Add, subtract (two low registers and immediate)</a>
0	!= 11		MOV, MOVS (register) — T2
1			<a href="#">Add, subtract, compare, move (one low register and immediate)</a>

**Add, subtract (three low registers)**

These instructions are under [Shift \(immediate\), add, subtract, move, and compare](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	S	Rm			Rn			Rd		

Decode fields S	Instruction Details
0	ADD, ADDS (register)
1	SUB, SUBS (register)

**Add, subtract (two low registers and immediate)**

These instructions are under [Shift \(immediate\), add, subtract, move, and compare](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	S	imm3			Rn			Rd		

Decode fields S	Instruction Details
0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

**Add, subtract, compare, move (one low register and immediate)**

These instructions are under [Shift \(immediate\), add, subtract, move, and compare](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	op			Rd			imm8						

Decode fields op	Instruction Details
00	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

**Data-processing (two low registers)**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	op				Rs			Rd		

Decode fields op	Instruction Details
0000	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) — logical shift left
0011	MOV, MOVS (register-shifted register) — logical shift right
0100	MOV, MOVS (register-shifted register) — arithmetic shift right
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) — rotate right
1000	TST (register)

Decode fields	Instruction Details
op	
1001	RSB, RSBS (immediate)
1010	CMP (register)
1011	CMN (register)
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

## Special data instructions and branch and exchange

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	op0									

Decode fields	Instruction details
op0	
11	<a href="#">Branch and exchange</a>
!= 11	<a href="#">Add, subtract, compare, move (two high registers)</a>

## Branch and exchange

These instructions are under [Special data instructions and branch and exchange](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	L		Rm		(0)	(0)	(0)	

Decode fields	Instruction Details
L	
0	BX
1	BLX (register)

## Add, subtract, compare, move (two high registers)

These instructions are under [Special data instructions and branch and exchange](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	!= 11	D		Rs					Rd	

op

The following constraints also apply to this encoding: op != 11 && op != 11

Decode fields			Instruction Details
op	D:Rd	Rs	
00	!= 1101	!= 1101	ADD, ADDS (register)
00		1101	ADD, ADDS (SP plus register) — T1
00	1101	!= 1101	ADD, ADDS (SP plus register) — T2
01			CMP (register)
10			MOV, MOVS (register)

**Load/store (register offset)**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	L	B	H	Rm			Rn			Rt		

Decode fields			Instruction Details
L	B	H	
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

**Load/store word/byte (immediate offset)**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	B	L	imm5					Rn			Rt		

Decode fields		Instruction Details
B	L	
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

**Load/store halfword (immediate offset)**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	L	imm5					Rn			Rt		

Decode fields		Instruction Details
L		
0		STRH (immediate)
1		LDRH (immediate)

**Load/store (SP-relative)**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	L	Rt			imm8							

Decode fields		Instruction Details
L		
0		STR (immediate)
1		LDR (immediate)

**Add PC/SP (immediate)**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	SP		Rd									imm8

Decode fields	Instruction Details
<b>SP</b>	
0	ADR
1	ADD, ADDS (SP plus immediate)

**Miscellaneous 16-bit instructions**

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
10	11		op0	op1	op2	op3									

op0	Decode fields op1	op2	op3	Instruction details	Architecture version
0000				<a href="#">Adjust SP (immediate)</a>	-
0010				<a href="#">Extend</a>	-
0110	00	0		SETPAN	FEAT_PAN
0110	00	1		UNALLOCATED	-
0110	01			<a href="#">Change Processor State</a>	-
0110	1x			UNALLOCATED	-
0111				UNALLOCATED	-
1000				UNALLOCATED	-
1010	10			<a href="#">HLT</a>	-
1010	!= 10			<a href="#">Reverse bytes</a>	-
1110				BKPT	-
1111			0000	<a href="#">Hints</a>	-
1111			!= 0000	IT	-
x0x1				CBNZ, CBZ	-
x10x				<a href="#">Push and Pop</a>	-

**Adjust SP (immediate)**

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	S							imm7

Decode fields	Instruction Details
<b>S</b>	
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

**Extend**

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	U	B		Rm			Rd	

Decode fields		Instruction Details
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
1	1	UXTB

## Change Processor State

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	op	flags				

Decode fields		Instruction Details
op	flags	
0		SETEND
1	0xxxx	CPS, CPSID, CPSIE — interrupt enable
1	1xxxx	CPS, CPSID, CPSIE — interrupt disable

## Reverse bytes

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	!= 10		Rm			Rd		
op															

The following constraints also apply to this encoding: op != 10 && op != 10

Decode fields		Instruction Details
op		
00		REV
01		REV16
11		REVSH

## Hints

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	hint				0	0	0	0

Decode fields		Instruction Details
hint		
0000		NOP
0001		YIELD
0010		WFE
0011		WFI
0100		SEV
0101		SEVL
011x		Reserved hint, behaves as NOP
1xxx		Reserved hint, behaves as NOP

## Push and Pop

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	L	1	0	P	register_list							

Decode fields	Instruction Details
L	
0	PUSH
1	POP

## Load/store multiple

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	L	Rn			register_list							

Decode fields	Instruction Details
L	
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

## Conditional branch, and Supervisor Call

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1101				op0											

Decode fields	Instruction details
op0	
111x	<a href="#">Exception generation</a>
!= 111x	B — T1

## Exception generation

These instructions are under [Conditional branch, and Supervisor Call](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	S	imm8							

Decode fields	Instruction Details
S	
0	UDF
1	SVC

## 32-bit

These instructions are under the [top-level](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111			op0					op1										op3													

The following constraints also apply to this encoding: op0<3:2> != 00

Decode fields			Instruction details
op0	op1	op3	
x11x			<a href="#">System register access, Advanced SIMD, and floating-point</a>
0100	xx0xx		<a href="#">Load/store multiple</a>
0100	xx1xx		<a href="#">Load/store dual, load/store exclusive, load-acquire/store-release, and table branch</a>
0101			<a href="#">Data-processing (shifted register)</a>
10xx		1	<a href="#">Branches and miscellaneous control</a>
10x0		0	<a href="#">Data-processing (modified immediate)</a>
10x1	xxxx0	0	<a href="#">Data-processing (plain binary immediate)</a>
10x1	xxxx1	0	UNALLOCATED
1100	1xxx0		<a href="#">Advanced SIMD element or structure load/store</a>
1100	!= 1xxx0		<a href="#">Load/store single</a>
1101	0xxxx		<a href="#">Data-processing (register)</a>
1101	10xxx		<a href="#">Multiply, multiply accumulate, and absolute difference</a>
1101	11xxx		<a href="#">Long multiply and divide</a>

## System register access, Advanced SIMD, and floating-point

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111	op0	11	op1													op2											op3				

Decode fields				Instruction details
op0	op1	op2	op3	
	0x	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			<a href="#">Advanced SIMD data-processing</a>
0	0x	1x		<a href="#">Advanced SIMD and System register load/store and 64-bit move</a>
0	10	1x	1	<a href="#">Advanced SIMD and System register 32-bit move</a>
0	10	10	0	<a href="#">Floating-point data-processing</a>
0	10	11	0	UNALLOCATED
1	!= 11	1x		<a href="#">Additional Advanced SIMD and floating-point instructions</a>

## Advanced SIMD data-processing

These instructions are under [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111			1111	op0																							op1				

Decode fields		Instruction details
op0	op1	
0		<a href="#">Advanced SIMD three registers of the same length</a>
1	0	<a href="#">Advanced SIMD two registers, or three registers of different lengths</a>
1	1	<a href="#">Advanced SIMD shifts and immediate generation</a>

## Advanced SIMD three registers of the same length

These instructions are under [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size	Vn	Vd	opc	N	Q	M	o1	Vm													

U	Decode fields		Q	o1	Instruction Details	Feature
	size	opc				
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) — T2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	-
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) — T1	-
		0011		1	VCGE (register) — T1	-
0	01	1100		0	SHA1P	-
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	<a href="#">VRSHL</a>	-
		0101		1	<a href="#">VQRSHL</a>	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	-
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-
1	0x	1110		0	VCGE (register) — T2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	-
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	-
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) — T2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) — T1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

## Advanced SIMD two registers, or three registers of different lengths

These instructions are under [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
111			op0		11111						op1							op2							op3			0						

Decode fields				Instruction details
op0	op1	op2	op3	
0	11			VEXT (byte elements)
1	11	0x		<a href="#">Advanced SIMD two registers misc</a>
1	11	10		VTBL, VTBX
1	11	11		<a href="#">Advanced SIMD duplicate (scalar)</a>
	!= 11		0	<a href="#">Advanced SIMD three registers of different lengths</a>
	!= 11		1	<a href="#">Advanced SIMD two registers and a scalar</a>

## Advanced SIMD two registers misc

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	opc1	Vd			0	opc2			Q	M	0	Vm							

Decode fields				Instruction Details	Feature
size	opc1	opc2	Q		
	00	0000		VREV64	-
	00	0001		VREV32	-

size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	-
	00	0110	1	AESD	-
	00	0111	0	AESMC	-
	00	0111	1	AESIMC	-
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VCGT (immediate #0)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	-
	10	0111	1	SHA256SU0	-
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-

size	Decode fields opc1	opc2	Q	Instruction Details	Feature
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRTE	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

### Advanced SIMD duplicate (scalar)

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	imm4			Vd			1	1	opc			Q	M	0	Vm					

Decode fields opc	Instruction Details
000	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

### Advanced SIMD three registers of different lengths

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	U	1	1	1	1	1	D	!= 11			Vn			Vd			opc			N	0	M	0	Vm						
size																																

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields U	opc	Instruction Details
	0000	VADDL
	0001	VADDW
	0010	VSUBL
0	0100	VADDHN
	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)
1	1001	UNALLOCATED
1	1011	UNALLOCATED

Decode fields	Instruction Details
U      opc	
1      1101	UNALLOCATED
1      1111	UNALLOCATED

## Advanced SIMD two registers and a scalar

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	!= 11	Vn				Vd				opc				N	1	M	0	Vm				
size																															

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields	Instruction Details	Feature
Q      opc		
	000x	VMLA (by scalar)
0	0011	VQDMLAL
	0010	VMLAL (by scalar)
0	0111	VQDMLSL
	010x	VMLS (by scalar)
0	1011	VQDMULL
	0110	VMLSL (by scalar)
	100x	VMUL (by scalar)
1	0011	UNALLOCATED
	1010	VMULL (by scalar)
1	0111	UNALLOCATED
	1100	VQDMULH
	1101	VQRDMULH
1	1011	UNALLOCATED
	1110	VQRDMLAH
	1111	VQRDMLSH

## Advanced SIMD shifts and immediate generation

These instructions are under [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111				11111					op0															1							

Decode fields	Instruction details
op0	
000xxxxxxxxxxx0	<a href="#">Advanced SIMD one register and modified immediate</a>
!= 000xxxxxxxxxxx0	<a href="#">Advanced SIMD two registers and shift amount</a>

## Advanced SIMD one register and modified immediate

These instructions are under [Advanced SIMD shifts and immediate generation](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			cmode			0	Q	op	1	imm4					

Decode fields		Instruction Details
cmode	op	
0xx0	0	VMOV (immediate) — T1
0xx0	1	VMVN (immediate) — T1
0xx1	0	VORR (immediate) — T1
0xx1	1	VBIC (immediate) — T1
10x0	0	VMOV (immediate) — T3
10x0	1	VMVN (immediate) — T2
10x1	0	VORR (immediate) — T2
10x1	1	VBIC (immediate) — T2
11xx	0	VMOV (immediate) — T4
110x	1	VMVN (immediate) — T3
1110	1	VMOV (immediate) — T5
1111	1	UNALLOCATED

### Advanced SIMD two registers and shift amount

These instructions are under [Advanced SIMD shifts and immediate generation](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm3H			imm3L		Vd			opc			L	Q	M	1	Vm						

The following constraints also apply to this encoding: imm3H:imm3L:Vd:opc:L != 000xxxxxxxxxxx0

Decode fields					Instruction Details
U	imm3H:L	imm3L	opc	Q	
	!= 0000		0000		VSHR
	!= 0000		0001		VSRA
	!= 0000	000	1010	0	VMOVL
	!= 0000		0010		VRSRHR
	!= 0000		0011		VRSRA
	!= 0000		0111		VQSHL, VQSHLU (immediate) — VQSHL
	!= 0000		1001	0	VQSHRN, VQSHRUN — VQSHRN
	!= 0000		1001	1	VQRSHRN, VQRSHRUN — VQRSHRN
	!= 0000		1010	0	VSHLL
	!= 0000		11xx		VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000		0101		VSHL (immediate)
0	!= 0000		1000	0	VSHRN
0	!= 0000		1000	1	VRSHRN
1	!= 0000		0100		VSRI
1	!= 0000		0101		VSLI
1	!= 0000		0110		VQSHL, VQSHLU (immediate) — VQSHLU
1	!= 0000		1000	0	VQSHRN, VQSHRUN — VQSHRUN
1	!= 0000		1000	1	VQRSHRN, VQRSHRUN — VQRSHRUN

### Advanced SIMD and System register load/store and 64-bit move

These instructions are under [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110110								op0								1	op1														

Decode fields		Instruction details
op0	op1	
00x0	0x	<a href="#">Advanced SIMD and floating-point 64-bit move</a>
00x0	11	<a href="#">System register 64-bit move</a>
!= 00x0	0x	<a href="#">Advanced SIMD and floating-point load/store</a>
!= 00x0	11	<a href="#">System register Load/Store</a>
	10	UNALLOCATED

### Advanced SIMD and floating-point 64-bit move

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	D	0	op	Rt2				Rt				1	0	size	opc2	M	o3	Vm					

Decode fields					Instruction Details
D	op	size	opc2	o3	
0					UNALLOCATED
1				0	UNALLOCATED
1		0x	00	1	UNALLOCATED
1			01		UNALLOCATED
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers
1			1x		UNALLOCATED
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers

### System register 64-bit move

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	D	0	L	Rt2				Rt			1	1	1	cp15	opc1			CRm					

Decode fields		Instruction Details
D	L	
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

### Advanced SIMD and floating-point load/store

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	L	Rn				Vd				1	0	size	imm8								

The following constraints also apply to this encoding: P:U:D:W != 00x0

P	U	W	Decode fields			imm8	Instruction Details	Feature
			L	Rn	size			
0	0	1					UNALLOCATED	-
0	1				0x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
0	1		0		11	XXXXXXXX0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
0	1		0		11	XXXXXXXX1	FSTMDBX, FSTMIAX — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
0	1		1		11	XXXXXXXX0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
0	1		1		11	XXXXXXXX1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After	-
1		0	0		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
1	0	1	0		11	XXXXXXXX0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
1	0	1	0		11	XXXXXXXX1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
1	0	1	1		11	XXXXXXXX0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
1	0	1	1		11	XXXXXXXX1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1		0	1	1111	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

## System register Load/Store

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	L	Rn				CRd			1	1	1	cp15			imm8						

The following constraints also apply to this encoding: P:U:D:W != 00x0

			Decode fields			
P:U:W	D	L	Rn	CRd	cp15	Instruction Details
!= 000				!= 0101	0	UNALLOCATED
!= 000	0	1	1111	0101	0	LDC (literal)

P:U:W	D	L	Decode fields Rn	CRd	cp15	Instruction Details
!= 000					1	UNALLOCATED
!= 000	1			0101	0	UNALLOCATED
0x1	0	0		0101	0	STC — post-indexed
0x1	0	1	!= 1111	0101	0	LDC (immediate) — post-indexed
010	0	0		0101	0	STC — unindexed
010	0	1	!= 1111	0101	0	LDC (immediate) — unindexed
1x0	0	0		0101	0	STC — offset
1x0	0	1	!= 1111	0101	0	LDC (immediate) — offset
1x1	0	0		0101	0	STC — pre-indexed
1x1	0	1	!= 1111	0101	0	LDC (immediate) — pre-indexed

## Advanced SIMD and System register 32-bit move

These instructions are under [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
11101110								op0								1	op1								1								

Decode fields op0 op1		Instruction details	Architecture version
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	<a href="#">Floating-point move special register</a>	-
	011	<a href="#">Advanced SIMD 8/16/32-bit element move/duplicate</a>	-
	10x	UNALLOCATED	-
	11x	<a href="#">System register 32-bit move</a>	-

## Floating-point move special register

These instructions are under [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	L	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields L	Instruction Details
0	VMSR
1	VMRS

## Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	0	1	1	1	0	opc1				L	Vn				Rt				1	0	1	1	N	opc2				1	(0)	(0)	(0)	(0)

Decode fields			Instruction Details
opc1	L	opc2	
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

## System register 32-bit move

These instructions are under [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1			L	CRn				Rt				1	1	1	cp15	opc2			1	CRm			

Decode fields		Instruction Details
L		
0		MCR
1		MRC

## Floating-point data-processing

These instructions are under [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11101110								op0								10					op1					0					

Decode fields		Instruction details
op0	op1	
1x11	1	<a href="#">Floating-point data-processing (two registers)</a>
1x11	0	<a href="#">Floating-point move immediate</a>
!= 1x11		<a href="#">Floating-point data-processing (three registers)</a>

## Floating-point data-processing (two registers)

These instructions are under [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	o1	opc2			Vd			1	0	size	o3	1	M	0	Vm					

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — single-precision to half-precision	-
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

### Floating-point move immediate

These instructions are under [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H			Vd			1	0	size	(0)	0	(0)	0	imm4L						

Decode fields size	Instruction Details	Feature
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	FEAT_FP16
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

### Floating-point data-processing (three registers)

These instructions are under [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	o0	D		o1	Vn				Vd			1	0	size	N	o2	M	0		Vm				

The following constraints also apply to this encoding: o0:D:o1 != 1x11

Decode fields o0:o1 size o2	Instruction Details		Feature
!= 111	00		UNALLOCATED
000	01	0	VMLA (floating-point) — half-precision scalar
000	01	1	VMLS (floating-point) — half-precision scalar
000	10	0	VMLA (floating-point) — single-precision scalar
000	10	1	VMLS (floating-point) — single-precision scalar
000	11	0	VMLA (floating-point) — double-precision scalar
000	11	1	VMLS (floating-point) — double-precision scalar
001	01	0	VNMLS — half-precision scalar

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

## Additional Advanced SIMD and floating-point instructions

These instructions are under [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111111						op0			op1								1	op2	op3		op4		op5								

The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			<a href="#">Advanced SIMD three registers of the same length extension</a>
100		0	!= 00	0	0	<a href="#">Floating-point conditional select</a>

101	00xxxx	0	!= 00		0	<a href="#">Floating-point minNum/maxNum</a>
101	110000	0	!= 00	1	0	<a href="#">Floating-point extraction and insertion</a>
101	111xxx	0	!= 00	1	0	<a href="#">Floating-point directed convert to integer</a>
10x		0	00			<a href="#">Advanced SIMD and floating-point multiply with accumulate</a>
10x		1	0x			<a href="#">Advanced SIMD and floating-point dot product</a>

### Advanced SIMD three registers of the same length extension

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	op1	D	op2	Vn			Vd			1	op3	0	op4	N	Q	M	U	Vm							

Decode fields						Instruction Details		Feature
op1	op2	op3	op4	Q	U			
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector		FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED		-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector		FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED		-
00	0x	0	0			UNALLOCATED		-
00	0x	0	1			UNALLOCATED		-
00	00	1	0	0	0	UNALLOCATED		-
00	00	1	0	0	1	UNALLOCATED		-
00	00	1	0	1	0	VMMLA		FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED		-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector		FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED		-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector		FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED		-
00	01	1	0			UNALLOCATED		-
00	01	1	1			UNALLOCATED		-
00	10	0	0		1	VFMAL (vector)		FEAT_FHM
00	10	0	1			UNALLOCATED		-
00	10	1	0	0		UNALLOCATED		-
00	10	1	0	1	0	VSMMLA		FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA		FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector		FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector		FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector		FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector		FEAT_DotProd
00	11	0	0		1	VFMA, VFMA, VFMA (BFloat16, vector)		FEAT_AA32BF16
00	11	0	1			UNALLOCATED		-
00	11	1	0			UNALLOCATED		-
00	11	1	1			UNALLOCATED		-
01	10	0	0		1	VFMSL (vector)		FEAT_FHM
01	10	0	1			UNALLOCATED		-
01	10	1	0	0		UNALLOCATED		-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	1			UNALLOCATED	-

### Floating-point conditional select

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc	Vn				Vd				1	0	!= 00	N	0	M	0	Vm					
size																															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields		Instruction Details	Feature
size			
01		VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10		VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11		VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

### Floating-point minNum/maxNum

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	!= 00	N	op	M	0	Vm				
size																															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields		Instruction Details	Feature
size	op		
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

## Floating-point extraction and insertion

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd		1	0	!= 00	op	1	M	0	Vm						
size																															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

## Floating-point directed convert to integer

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	o1	RM				Vd			1	0	!= 00	op	1	M	0			Vm	
size																															

The following constraints also apply to this encoding: size != 00 && size != 00

o1	Decode fields RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-

Decode fields			Instruction Details				Feature
o1	RM	size	op				
1	11	11		VCVTM (floating-point) — double-precision scalar			-

### Advanced SIMD and floating-point multiply with accumulate

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2	Vn			Vd			1	0	0	0	N	Q	M	U	Vm						

Decode fields				Instruction Details				Feature
op1	op2	Q	U					
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point			FEAT_FCMA	
0	00		1	VFMAL (by scalar)			FEAT_FHM	
0	01		1	VFMSL (by scalar)			FEAT_FHM	
0	10		1	UNALLOCATED			-	
0	11		1	VFMAb, VFMAb (BFloat16, by scalar)			FEAT_AA32BF16	
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point			FEAT_FCMA	
1			1	UNALLOCATED			-	
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point			FEAT_FCMA	

### Advanced SIMD and floating-point dot product

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	0	op1	D	op2	Vn					Vd					1	1	0	op4	N	Q	M	U	Vm				

Decode fields					Instruction Details				Feature
op1	op2	op4	Q	U					
0	00	0			UNALLOCATED			-	
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector			FEAT_AA32BF16	
0	00	1		1	UNALLOCATED			-	
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector			FEAT_AA32BF16	
0	01	0			UNALLOCATED			-	
0	10	0			UNALLOCATED			-	
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector			FEAT_DotProd	
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector			FEAT_DotProd	
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector			FEAT_DotProd	
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector			FEAT_DotProd	
0	11				UNALLOCATED			-	
1		0			UNALLOCATED			-	
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector			FEAT_AA32I8MM	
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector			FEAT_AA32I8MM	
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector			FEAT_AA32I8MM	
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector			FEAT_AA32I8MM	
1	01	1			UNALLOCATED			-	
1	1x	1			UNALLOCATED			-	

## Load/store multiple

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	opc		0	W	L	Rn			P	M	register_list														

Decode fields		Instruction Details
opc	L	
00	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — T1
00	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — T1
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — T2
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — T2

## Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110100							op0			op1	op2			op3																	

The following constraints also apply to this encoding: op0<1> == 1

Decode fields				Instruction details
op0	op1	op2	op3	
0010				<a href="#">Load/store exclusive</a>
0110	0		000	UNALLOCATED
0110	1		000	TBB, TBH
0110			01x	<a href="#">Load/store exclusive byte/half/dual</a>
0110			1xx	<a href="#">Load-acquire / Store-release</a>
0x11		!= 1111		<a href="#">Load/store dual (immediate, post-indexed)</a>
1x10		!= 1111		<a href="#">Load/store dual (immediate)</a>
1x11		!= 1111		<a href="#">Load/store dual (immediate, pre-indexed)</a>
!= 0xx0		1111		LDRD (literal)

## Load/store exclusive

These instructions are under [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	L	Rn			Rt			Rd			imm8										

Decode fields		Instruction Details
L		
0		STREX
1		LDREX

## Load/store exclusive byte/half/dual

These instructions are under [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	L	Rn			Rt			Rt2			0 1		sz		Rd						

Decode fields		Instruction Details
L	sz	
0	00	STREXB
0	01	STREXH
0	10	UNALLOCATED
0	11	STREXD
1	00	LDREXB
1	01	LDREXH
1	10	UNALLOCATED
1	11	LDREXD

## Load-acquire / Store-release

These instructions are under [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	L	Rn				Rt				Rt2			1	op	sz		Rd				

Decode fields			Instruction Details
L	op	sz	
0	0	00	STLB
0	0	01	STLH
0	0	10	STL
0	0	11	UNALLOCATED
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA
1	0	11	UNALLOCATED
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

## Load/store dual (immediate, post-indexed)

These instructions are under [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	U	1	1	L	!= 1111			Rt			Rt2			imm8										
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
L		
0		STRD (immediate)
1		LDRD (immediate)

**Load/store dual (immediate)**

These instructions are under [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	U	1	0	L	!= 1111				Rt				Rt2				imm8							
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

**Load/store dual (immediate, pre-indexed)**

These instructions are under [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	U	1	1	L	!= 1111				Rt				Rt2				imm8							
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

**Data-processing (shifted register)**

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	op1				S	Rn				(0)	imm3				Rd				imm2	stype	Rm				

op1	S	Rn	Decode fields imm3:imm2:stype	Rd	Instruction Details
0000	0		!= 0000011		AND, ANDS (register) — AND, shift or rotate by value
0000	0		0000011		AND, ANDS (register) — AND, rotate right with extend
0000	1		!= 0000011	!= 1111	AND, ANDS (register) — ANDS, shift or rotate by value
0000	1		!= 0000011	1111	TST (register) — shift or rotate by value
0000	1		0000011	!= 1111	AND, ANDS (register) — ANDS, rotate right with extend
0000	1		0000011	1111	TST (register) — rotate right with extend
0001			!= 0000011		BIC, BICS (register) — shift or rotate by value
0001			0000011		BIC, BICS (register) — rotate right with extend
0010	0	!= 1111	!= 0000011		ORR, ORRS (register) — ORR, shift or rotate by value
0010	0	!= 1111	0000011		ORR, ORRS (register) — ORR, rotate right with extend
0010	0	1111	!= 0000011		MOV, MOVS (register) — MOV, shift or rotate by value

op1	S	Decode fields		Rd	Instruction Details
		Rn	imm3:imm2:type		
0010	0	1111	0000011		MOV, MOVS (register) — MOV, rotate right with extend
0010	1	!= 1111	!= 0000011		ORR, ORRS (register) — ORRS, shift or rotate by value
0010	1	!= 1111	0000011		ORR, ORRS (register) — ORRS, rotate right with extend
0010	1	1111	!= 0000011		MOV, MOVS (register) — MOVS, shift or rotate by value
0010	1	1111	0000011		MOV, MOVS (register) — MOVS, rotate right with extend
0011	0	!= 1111	!= 0000011		ORN, ORNS (register) — ORN, shift or rotate by value
0011	0	!= 1111	0000011		ORN, ORNS (register) — ORN, rotate right with extend
0011	0	1111	!= 0000011		MVN, MVNS (register) — MVN, shift or rotate by value
0011	0	1111	0000011		MVN, MVNS (register) — MVN, rotate right with extend
0011	1	!= 1111	!= 0000011		ORN, ORNS (register) — ORNS, shift or rotate by value
0011	1	!= 1111	0000011		ORN, ORNS (register) — ORNS, rotate right with extend
0011	1	1111	!= 0000011		MVN, MVNS (register) — MVNS, shift or rotate by value
0011	1	1111	0000011		MVN, MVNS (register) — MVNS, rotate right with extend
0100	0		!= 0000011		EOR, EORS (register) — EOR, shift or rotate by value
0100	0		0000011		EOR, EORS (register) — EOR, rotate right with extend
0100	1		!= 0000011	!= 1111	EOR, EORS (register) — EORS, shift or rotate by value
0100	1		!= 0000011	1111	TEQ (register) — shift or rotate by value
0100	1		0000011	!= 1111	EOR, EORS (register) — EORS, rotate right with extend
0100	1		0000011	1111	TEQ (register) — rotate right with extend
0101					UNALLOCATED
0110	0		xxxxx00		PKHBT, PKHTB — PKHBT
0110	0		xxxxx01		UNALLOCATED
0110	0		xxxxx10		PKHBT, PKHTB — PKHTB
0110	0		xxxxx11		UNALLOCATED
0111					UNALLOCATED
1000	0	!= 1101	!= 0000011		ADD, ADDS (register) — ADD, shift or rotate by value
1000	0	!= 1101	0000011		ADD, ADDS (register) — ADD, rotate right with extend
1000	0	1101	!= 0000011		ADD, ADDS (SP plus register) — ADD, shift or rotate by value
1000	0	1101	0000011		ADD, ADDS (SP plus register) — ADD, rotate right with extend
1000	1		!= 0000011	1111	CMN (register) — shift or rotate by value

op1	S	Decode fields		Rd	Instruction Details
		Rn	imm3:imm2:stype		
1000	1	!= 1101	!= 0000011	!= 1111	ADD, ADDS (register) — ADDS, shift or rotate by value
1000	1	!= 1101	0000011	!= 1111	ADD, ADDS (register) — ADDS, rotate right with extend
1000	1		0000011	1111	CMN (register) — rotate right with extend
1000	1	1101	!= 0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
1000	1	1101	0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
1001					UNALLOCATED
1010			!= 0000011		ADC, ADCS (register) — shift or rotate by value
1010			0000011		ADC, ADCS (register) — rotate right with extend
1011			!= 0000011		SBC, SBCS (register) — shift or rotate by value
1011			0000011		SBC, SBCS (register) — rotate right with extend
1100					UNALLOCATED
1101	0	!= 1101	!= 0000011		SUB, SUBS (register) — SUB, shift or rotate by value
1101	0	!= 1101	0000011		SUB, SUBS (register) — SUB, rotate right with extend
1101	0	1101	!= 0000011		SUB, SUBS (SP minus register) — SUB, shift or rotate by value
1101	0	1101	0000011		SUB, SUBS (SP minus register) — SUB, rotate right with extend
1101	1		!= 0000011	1111	CMP (register) — shift or rotate by value
1101	1	!= 1101	!= 0000011	!= 1111	SUB, SUBS (register) — SUBS, shift or rotate by value
1101	1	!= 1101	0000011	!= 1111	SUB, SUBS (register) — SUBS, rotate right with extend
1101	1		0000011	1111	CMP (register) — rotate right with extend
1101	1	1101	!= 0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
1101	1	1101	0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
1110			!= 0000011		RSB, RSBS (register) — shift or rotate by value
1110			0000011		RSB, RSBS (register) — rotate right with extend
1111					UNALLOCATED

## Branches and miscellaneous control

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
11110					op0	op1					op2					1					op3					op4					op5				

op0	op1	Decode fields				op5	Instruction details
		op2	op3	op4			
0	1110	0x	0x0		0		MSR (register)
0	1110	0x	0x0		1		MSR (Banked register)
0	1110	10	0x0	000			<a href="#">Hints</a>
0	1110	10	0x0	!= 000			<a href="#">Change processor state</a>

0	1110	11	0x0			<a href="#">Miscellaneous system</a>
0	1111	00	0x0			BXJ
0	1111	01	0x0			<a href="#">Exception return</a>
0	1111	1x	0x0		0	MRS
0	1111	1x	0x0		1	MRS (Banked register)
1	1110	00	000			<a href="#">DCPS</a>
1	1110	00	010			UNALLOCATED
1	1110	01	0x0			UNALLOCATED
1	1110	1x	0x0			UNALLOCATED
1	1111	0x	0x0			UNALLOCATED
1	1111	1x	0x0			<a href="#">Exception generation</a>
	!= 111x		0x0			B — T3
			0x1			B — T4
			1x0			BL, BLX (immediate) — T2
			1x1			BL, BLX (immediate) — T1

## Hints

These instructions are under [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0		hint					option	

Decode fields hint	option	Instruction Details	Feature
0000	0000	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP	-
0000	1xxx	Reserved hint, behaves as NOP	-
0001	0000	ESB	FEAT_RAS
0001	0001	Reserved hint, behaves as NOP	-
0001	0010	<a href="#">TSB CSYNC</a>	FEAT_TRF
0001	0011	Reserved hint, behaves as NOP	-
0001	0100	CSDB	-
0001	0101	Reserved hint, behaves as NOP	-
0001	011x	Reserved hint, behaves as NOP	-
0001	1xxx	Reserved hint, behaves as NOP	-
001x		Reserved hint, behaves as NOP	-
01xx		Reserved hint, behaves as NOP	-
10xx		Reserved hint, behaves as NOP	-
110x		Reserved hint, behaves as NOP	-
1110		Reserved hint, behaves as NOP	-
1111		DBG	-

## Change processor state

These instructions are under [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod	M	A	I	F	mode					

The following constraints also apply to this encoding: imod:M != 000

Decode fields		Instruction Details
imod	M	
00	1	CPS, CPSID, CPSIE — change mode
01		UNALLOCATED
10		CPS, CPSID, CPSIE — interrupt enable and change mode
11		CPS, CPSID, CPSIE — interrupt disable and change mode

## Miscellaneous system

These instructions are under [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	opc				option			

Decode fields		Instruction Details	Feature
opc	option		
000x		UNALLOCATED	—
0010		CLREX	—
0011		UNALLOCATED	—
0100	!= 0x00	<a href="#">DSB</a>	—
0100	0000	SSBB	—
0100	0100	PSSBB	—
0101		DMB	—
0110		ISB	—
0111		<a href="#">SB</a>	FEAT_SB
1xxx		UNALLOCATED	—

## Exception return

These instructions are under [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1	Rn				1	0	(0)	0	(1)	(1)	(1)	(1)	imm8							

Decode fields	Instruction Details
Rn:imm8	
!= 111000000000	SUB, SUBS (immediate)
111000000000	ERET

## DCPS

These instructions are under [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	imm4				1	0	0	0	imm10								opt			

Decode fields		opt	Instruction Details
imm4	imm10		
!= 1111			UNALLOCATED
1111	!= 0000000000		UNALLOCATED

Decode fields		opt	Instruction Details
imm4	imm10		
1111	0000000000	00	UNALLOCATED
1111	0000000000	01	DCPS1
1111	0000000000	10	<a href="#">DCPS2</a>
1111	0000000000	11	<a href="#">DCPS3</a>

### Exception generation

These instructions are under [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	o1	imm4				1	0	o2	0	imm12											

Decode fields		Instruction Details
o1	o2	
0	0	<a href="#">HVC</a>
0	1	UNALLOCATED
1	0	<a href="#">SMC</a>
1	1	UDF

### Data-processing (modified immediate)

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0					op1	S			Rn	0			imm3			Rd									imm8

Decode fields			Rd	Instruction Details
op1	S	Rn		
0000	0			AND, ANDS (immediate) — AND
0000	1		!= 1111	AND, ANDS (immediate) — ANDS
0000	1		1111	TST (immediate)
0001				BIC, BICS (immediate)
0010	0	!= 1111		ORR, ORRS (immediate) — ORR
0010	0	1111		MOV, MOVS (immediate) — MOV
0010	1	!= 1111		ORR, ORRS (immediate) — ORRS
0010	1	1111		MOV, MOVS (immediate) — MOVS
0011	0	!= 1111		ORN, ORNS (immediate) — not flag setting
0011	0	1111		MVN, MVNS (immediate) — MVN
0011	1	!= 1111		ORN, ORNS (immediate) — flag setting
0011	1	1111		MVN, MVNS (immediate) — MVNS
0100	0			EOR, EORS (immediate) — EOR
0100	1		!= 1111	EOR, EORS (immediate) — EORS
0100	1		1111	TEQ (immediate)
0101				UNALLOCATED
011x				UNALLOCATED
1000	0	!= 1101		ADD, ADDS (immediate) — ADD
1000	0	1101		ADD, ADDS (SP plus immediate) — ADD
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) — ADDS
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) — ADDS
1000	1		1111	CMN (immediate)

Decode fields				Instruction Details
op1	S	Rn	Rd	
1001				UNALLOCATED
1010				ADC, ADCS (immediate)
1011				SBC, SBCS (immediate)
1100				UNALLOCATED
1101	0	!= 1101		SUB, SUBS (immediate) — SUB
1101	0	1101		SUB, SUBS (SP minus immediate) — SUB
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) — SUBS
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) — SUBS
1101	1		1111	CMP (immediate)
1110				RSB, RSBS (immediate)
1111				UNALLOCATED

## Data-processing (plain binary immediate)

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		1	op0			op1	0					0															

Decode fields		Instruction details
op0	op1	
0	0x	<a href="#">Data-processing (simple immediate)</a>
0	10	<a href="#">Move Wide (16-bit immediate)</a>
0	11	UNALLOCATED
1		<a href="#">Saturate, Bitfield</a>

## Data-processing (simple immediate)

These instructions are under [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	o1	0	o2	0			Rn	0			imm3			Rd								imm8		

Decode fields			Instruction Details
o1	o2	Rn	
0	0	!= 11x1	ADD, ADDS (immediate)
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR — T3
0	1		UNALLOCATED
1	0		UNALLOCATED
1	1	!= 11x1	SUB, SUBS (immediate)
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR — T2

## Move Wide (16-bit immediate)

These instructions are under [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	o1	1	0	0			imm4	0			imm3			Rd								imm8		

Decode fields	Instruction Details
<b>o1</b>	
0	MOV, MOVS (immediate)
1	MOVT

## Saturate, Bitfield

These instructions are under [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1		op1		0			Rn		0		imm3			Rd		imm2	(0)			widthm1				

Decode fields	Instruction Details
<b>op1</b>	
<b>Rn</b>	
<b>imm3:imm2</b>	
000	SSAT — logical shift left
001	!= 00000 SSAT — arithmetic shift right
001	00000 SSAT16
010	<a href="#">SBFX</a>
011	!= 1111 <a href="#">BFI</a>
011	1111 <a href="#">BFC</a>
100	USAT — logical shift left
101	!= 00000 USAT — arithmetic shift right
101	00000 USAT16
110	<a href="#">UBFX</a>
111	UNALLOCATED

## Advanced SIMD element or structure load/store

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11111001								op0		0										op1											

Decode fields	Instruction details
<b>op0</b>	
<b>op1</b>	
0	<a href="#">Advanced SIMD load/store multiple structures</a>
1	11 <a href="#">Advanced SIMD load single structure to all lanes</a>
1	!= 11 <a href="#">Advanced SIMD load/store single structure to one lane</a>

## Advanced SIMD load/store multiple structures

These instructions are under [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	L	0	Rn				Vd				itype				size		align		Rm			

Decode fields	Instruction Details
<b>L</b>	
<b>itype</b>	
<b>Rm</b>	
0	000x != 11x1 VST4 (multiple 4-element structures)
0	000x 1101 VST4 (multiple 4-element structures)
0	000x 1111 VST4 (multiple 4-element structures)
0	0010 != 11x1 VST1 (multiple single elements)
0	0010 1101 VST1 (multiple single elements)
0	0010 1111 VST1 (multiple single elements)

L	Decode fields		Instruction Details
	itype	Rm	
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

### Advanced SIMD load single structure to all lanes

These instructions are under [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn				Vd				1	1	N	size	T	a	Rm					

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		!= 11x1	VLD1 (single element to all lanes)
1	00		1101	VLD1 (single element to all lanes)
1	00		1111	VLD1 (single element to all lanes)
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

### Advanced SIMD load/store single structure to one lane

These instructions are under [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn			Vd			!= 11	N	index_align			Rm								
																size															

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)

Decode fields				Instruction Details
L	size	N	Rm	
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)

Decode fields				Instruction Details
L	size	N	Rm	
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

## Load/store single

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111100								op0				op1		op2		op3															

The following constraints also apply to this encoding: op0<1>:op1 != 10

Decode fields				Instruction details
op0	op1	op2	op3	
00		!= 1111	000000	<a href="#">Load/store, unsigned (register offset)</a>
00		!= 1111	000001	UNALLOCATED
00		!= 1111	00001x	UNALLOCATED
00		!= 1111	0001xx	UNALLOCATED
00		!= 1111	001xxx	UNALLOCATED
00		!= 1111	01xxxx	UNALLOCATED
00		!= 1111	10x0xx	UNALLOCATED
00		!= 1111	10x1xx	<a href="#">Load/store, unsigned (immediate, post-indexed)</a>
00		!= 1111	1100xx	<a href="#">Load/store, unsigned (negative immediate)</a>
00		!= 1111	1110xx	<a href="#">Load/store, unsigned (unprivileged)</a>
00		!= 1111	11x1xx	<a href="#">Load/store, unsigned (immediate, pre-indexed)</a>
01		!= 1111		<a href="#">Load/store, unsigned (positive immediate)</a>
0x		1111		<a href="#">Load, unsigned (literal)</a>
10	1	!= 1111	000000	<a href="#">Load/store, signed (register offset)</a>
10	1	!= 1111	000001	UNALLOCATED
10	1	!= 1111	00001x	UNALLOCATED
10	1	!= 1111	0001xx	UNALLOCATED
10	1	!= 1111	001xxx	UNALLOCATED
10	1	!= 1111	01xxxx	UNALLOCATED
10	1	!= 1111	10x0xx	UNALLOCATED
10	1	!= 1111	10x1xx	<a href="#">Load/store, signed (immediate, post-indexed)</a>
10	1	!= 1111	1100xx	<a href="#">Load/store, signed (negative immediate)</a>
10	1	!= 1111	1110xx	<a href="#">Load/store, signed (unprivileged)</a>
10	1	!= 1111	11x1xx	<a href="#">Load/store, signed (immediate, pre-indexed)</a>
11	1	!= 1111		<a href="#">Load/store, signed (positive immediate)</a>
1x	1	1111		<a href="#">Load, signed (literal)</a>

## Load/store, unsigned (register offset)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	size	L	!= 1111	Rt				0	0	0	0	0	0	imm2				Rm					

Rn

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD, PLDW (register) — preload read
01	0		STRH (register)
01	1	!= 1111	LDRH (register)
01	1	1111	PLD, PLDW (register) — preload write
10	0		STR (register)
10	1		LDR (register)
11			UNALLOCATED

### Load/store, unsigned (immediate, post-indexed)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	size	L	!= 1111		Rt			1	0	U	1												imm8

Rn

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11		UNALLOCATED

### Load/store, unsigned (negative immediate)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	size	L	!= 1111		Rt			1	1	0	0												imm8

Rn

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read
01	0		STRH (immediate)

Decode fields			Instruction Details
size	L	Rt	
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) — preload write
10	0		STR (immediate)
10	1		LDR (immediate)
11			UNALLOCATED

**Load/store, unsigned (unprivileged)**

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	size	L	!= 1111	Rt				1	1	1	0	imm8											
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11		UNALLOCATED

**Load/store, unsigned (immediate, pre-indexed)**

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	size	L	!= 1111		Rt			1	1	U	1												imm8
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11		UNALLOCATED

**Load/store, unsigned (positive immediate)**

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	size	L	!= 1111				Rt				imm12												
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read
01	0		STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) — preload write
10	0		STR (immediate)
10	1		LDR (immediate)

### Load, unsigned (literal)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	size	L	1	1	1	1		Rt				imm12											

Decode fields			Instruction Details
size	L	Rt	
0x	1	1111	PLD (literal)
00	1	!= 1111	LDRB (literal)
01	1	!= 1111	LDRH (literal)
10	1		LDR (literal)
11			UNALLOCATED

### Load/store, signed (register offset)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1	!= 1111				Rt				0 0 0 0 0 0				imm2				Rm				
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	Rt		
00	!= 1111		LDRSB (register)
00	1111		PLI (register)
01	!= 1111		LDRSH (register)
01	1111		Reserved hint, behaves as NOP
1x			UNALLOCATED

### Load/store, signed (immediate, post-indexed)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1	!= 1111	Rt				1	0	U	1	imm8											
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

### Load/store, signed (negative immediate)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1	!= 1111	Rt				1	1	0	0	imm8											
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Rt	Instruction Details
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

### Load/store, signed (unprivileged)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 1 1 1 0 0 1 0									size		1		!= 1111			Rt			1 1 1 0				imm8								
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSBT
01	LDRSHT
1x	UNALLOCATED

### Load/store, signed (immediate, pre-indexed)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1	!= 1111	Rt				1	1	U	1	imm8											
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

### Load/store, signed (positive immediate)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	1	1	size	1	!= 1111					Rt					imm12											
Rn																																

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Rt	Instruction Details
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP

### Load, signed (literal)

These instructions are under [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	size	1	1	1	1	1		Rt	imm12														

Decode fields size	Rt	Instruction Details
00	!= 1111	LDRSB (literal)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (literal)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

### Data-processing (register)

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11111010								op0								op1						op2									

Decode fields op0	op1	op2	Instruction details
0	1111	0000	MOV, MOVS (register-shifted register) — T2, Flag setting
0	1111	0001	UNALLOCATED
0	1111	001x	UNALLOCATED
0	1111	01xx	UNALLOCATED
0	1111	1xxx	<a href="#">Register extends</a>
1	1111	0xxx	<a href="#">Parallel add-subtract</a>
1	1111	10xx	<a href="#">Data-processing (two source registers)</a>

1	1111	11xx	UNALLOCATED
	!= 1111		UNALLOCATED

## Register extends

These instructions are under [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	op1	U				Rn		1	1	1	1			Rd		1	(0)	rotate			Rm		

Decode fields			Instruction Details
op1	U	Rn	
00	0	!= 1111	SXTAH
00	0	1111	SXTH
00	1	!= 1111	UXTAH
00	1	1111	UXTH
01	0	!= 1111	SXTAB16
01	0	1111	SXTB16
01	1	!= 1111	UXTAB16
01	1	1111	UXTB16
10	0	!= 1111	SXTAB
10	0	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11			UNALLOCATED

## Parallel add-subtract

These instructions are under [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	op1					Rn		1	1	1	1			Rd		0	U	H	S			Rm	

Decode fields				Instruction Details
op1	U	H	S	
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	UNALLOCATED
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
000	1	1	1	UNALLOCATED
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	UNALLOCATED
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	UNALLOCATED
010	0	0	0	SASX

Decode fields				Instruction Details
op1	U	H	S	
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	UNALLOCATED
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	UNALLOCATED
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	0	1	1	UNALLOCATED
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	UNALLOCATED
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	UNALLOCATED
101	1	0	0	USUB16
101	1	0	1	UQSUB16
101	1	1	0	UHSUB16
101	1	1	1	UNALLOCATED
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	UNALLOCATED
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	UNALLOCATED
111				UNALLOCATED

### Data-processing (two source registers)

These instructions are under [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	op1			Rn				1	1	1	1	Rd			1	0	op2		Rm				

Decode fields		Instruction Details
op1	op2	
000	00	QADD
000	01	QDADD
000	10	QSUB
000	11	QDSUB
001	00	REV
001	01	REV16
001	10	RBIT

Decode fields		Instruction Details
op1	op2	
001	11	REVSH
010	00	SEL
010	01	UNALLOCATED
010	1x	UNALLOCATED
011	00	CLZ
011	01	UNALLOCATED
011	1x	UNALLOCATED
100	00	CRC32 — CRC32B
100	01	CRC32 — CRC32H
100	10	CRC32 — CRC32W
100	11	CONSTRAINED UNPREDICTABLE
101	00	CRC32C — CRC32CB
101	01	CRC32C — CRC32CH
101	10	CRC32C — CRC32CW
101	11	CONSTRAINED UNPREDICTABLE
11x		UNALLOCATED

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

## Multiply, multiply accumulate, and absolute difference

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111110110																				op0											

Decode fields		Instruction details
op0		
00		<a href="#">Multiply and absolute difference</a>
01		UNALLOCATED
1x		UNALLOCATED

## Multiply and absolute difference

These instructions are under [Multiply, multiply accumulate, and absolute difference](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	op1			Rn			Ra			Rd			0 0		op2		Rm						

Decode fields			Instruction Details
op1	Ra	op2	
000	!= 1111	00	MLA, MLAS
000		01	MLS
000		1x	UNALLOCATED
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT — SMLABB
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT — SMLABT
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT — SMLATB
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT — SMLATT

Decode fields			Instruction Details
op1	Ra	op2	
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT — SMULBB
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT — SMULBT
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT — SMULTB
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT — SMULTT
010	!= 1111	00	SMLAD, SMLADX — SMLAD
010	!= 1111	01	SMLAD, SMLADX — SMLADX
010		1x	UNALLOCATED
010	1111	00	SMUAD, SMUADX — SMUAD
010	1111	01	SMUAD, SMUADX — SMUADX
011	!= 1111	00	SMLAWB, SMLAWT — SMLAWB
011	!= 1111	01	SMLAWB, SMLAWT — SMLAWT
011		1x	UNALLOCATED
011	1111	00	SMULWB, SMULWT — SMULWB
011	1111	01	SMULWB, SMULWT — SMULWT
100	!= 1111	00	SMLSD, SMLSDX — SMLSD
100	!= 1111	01	SMLSD, SMLSDX — SMLSDX
100		1x	UNALLOCATED
100	1111	00	SMUSD, SMUSDX — SMUSD
100	1111	01	SMUSD, SMUSDX — SMUSDX
101	!= 1111	00	SMMLA, SMMLAR — SMMLA
101	!= 1111	01	SMMLA, SMMLAR — SMMLAR
101		1x	UNALLOCATED
101	1111	00	SMMUL, SMMULR — SMMUL
101	1111	01	SMMUL, SMMULR — SMMULR
110		00	SMMLS, SMMLSR — SMMLS
110		01	SMMLS, SMMLSR — SMMLSR
110		1x	UNALLOCATED
111	!= 1111	00	USADA8
111		01	UNALLOCATED
111		1x	UNALLOCATED
111	1111	00	USAD8

## Long multiply and divide

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	op1			Rn			RdLo			RdHi			op2			Rm							

Decode fields		Instruction Details
op1	op2	
000	!= 0000	UNALLOCATED
000	0000	SMULL, SMULLS
001	!= 1111	UNALLOCATED
001	1111	SDIV
010	!= 0000	UNALLOCATED
010	0000	UMULL, UMULLS
011	!= 1111	UNALLOCATED
011	1111	UDIV

Decode fields		Instruction Details
op1	op2	
100	0000	SMLAL, SMLALS
100	0001	UNALLOCATED
100	001x	UNALLOCATED
100	01xx	UNALLOCATED
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBB
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBT
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTB
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTT
100	1100	SMLALD, SMLALDX — SMLALD
100	1101	SMLALD, SMLALDX — SMLALDX
100	111x	UNALLOCATED
101	0xxx	UNALLOCATED
101	10xx	UNALLOCATED
101	1100	SMLSLD, SMLSLDX — SMLSLD
101	1101	SMLSLD, SMLSLDX — SMLSLDX
101	111x	UNALLOCATED
110	0000	UMLAL, UMLALS
110	0001	UNALLOCATED
110	001x	UNALLOCATED
110	010x	UNALLOCATED
110	0110	UMAAL
110	0111	UNALLOCATED
110	1xxx	UNALLOCATED
111		UNALLOCATED

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12\_to\_suppress\_diffs\_in\_2022\_03\_RC1 ; Build timestamp: 2022-03-29T10:46:11

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

**Pseudocodes**



```

// AArch32.AT()
// =====
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, ATAccess ataccess)
    TranslationStage stage = stage_in;
    SecurityState ss;
    Regime regime;
    boolean eae;

    // ATS1Hx instructions
    if el == EL2 then
        regime = Regime_EL2;
        eae = TRUE;
        ss = SS_NonSecure;

    // ATS1Cxx instructions
    elsif stage == TranslationStage_1 || (stage == TranslationStage_12 && !HaveEL(EL2)) then
        stage = TranslationStage_1;
        ss = SecurityStateAtEL(PSTATE.EL);
        regime = if ss == SS_Secure && ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL10;
        eae = TTBCR.EAE == '1';

    // ATS12NS0xx instructions
    else
        regime = Regime_EL10;
        eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR.EAE == '1';
        ss = SS_NonSecure;

    AddressDescriptor addrdesc;
    SDFType sdftype;
    aligned = TRUE;
    ispriv = el != EL0;
    supersection = '0';
    iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;

    // Prepare fault fields in case a fault is detected
    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    if eae then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, vaddress, acctype, aligned,
                                                    iswrite, ispriv);
    else
        (fault, addrdesc, sdftype) = AArch32.S1TranslateSD(fault, regime, ss, vaddress, acctype,
                                                            aligned, iswrite, ispriv);
        supersection = if sdftype == SDFType_Supersection then '1' else '0';

    // ATS12NS0xx instructions
    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        s2fslwalk = FALSE;
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fslwalk, acctype, aligned,
                                                iswrite, ispriv);

    if fault.statuscode != Fault_None then
        // Take exception when External abort occurs on translation table walk
        if (IsExternalAbort(fault) || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL1
                                        && EL2Enabled() && fault.s2fslwalk)) then
            PAR = bits(64) UNKNOWN;
            AArch32.Abort(vaddress, fault);

    addrdesc.fault = fault;

    if (eae || (stage == TranslationStage_12 && (HCR.VM == '1' || HCR.DC == '1'))
        || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL2)) then
        AArch32.EncodePARLD(addrdesc, ss);
    else

```

```

    AArch32.EncodePARSD(addrdesc, supersection, ss);
return;

```

## Library pseudocode for aarch32/at/AArch32.EncodePARLD

```

// AArch32.EncodePARLD()
// =====
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

    if !IsFault(addrdesc) then
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        PAR.F      = '0';
        PAR.SH     = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR.NS     = ns;
        PAR<10>    = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";           // IMPDEF
        PAR.LPAE   = '1';
        PAR.PA     = addrdesc.paddress.address<39:12>;
        PAR.ATTR   = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
    else
        PAR.F      = '1';
        PAR.FST    = AArch32.PARFaultStatusLD(addrdesc.fault);
        PAR.S2WLK  = if addrdesc.fault.s2fslwalk then '1' else '0';
        PAR.FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
        PAR.LPAE   = '1';
        PAR<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";         // IMPDEF
    return;

```

## Library pseudocode for aarch32/at/AArch32.EncodePARSD

```
// AArch32.EncodePARSD()
// =====
// Returns 32-bit format PAR on address translation instruction.

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, SecurityState ss)
    AddressDescriptor addrdesc = addrdesc_in;
    if !IsFault(addrdesc) then
        if (addrdesc.memattrs.memtype == MemType_Device ||
            (addrdesc.memattrs.inner.attrs == MemAttr_NC &&
             addrdesc.memattrs.outer.attrs == MemAttr_NC)) then
            addrdesc.memattrs.shareability = Shareability_OSH;
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        bits(2) sh = if addrdesc.memattrs.shareability != Shareability_NSH then '01' else '00';
        PAR.F      = '0';
        PAR.SS     = supersection;
        PAR.Outer  = AArch32.ReportedOuterAttrs(AArch32.PAROuterAttrs(addrdesc.memattrs));
        PAR.Inner  = AArch32.ReportedInnerAttrs(AArch32.PARInnerAttrs(addrdesc.memattrs));
        PAR.SH     = ReportedPARShareability(sh);
        PAR<8>     = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";           // IMPDEF
        PAR.NS     = ns;
        PAR.NOS    = if addrdesc.memattrs.shareability == Shareability_OSH then '0' else '1';
        PAR.LPAE   = '0';
        PAR.PA     = addrdesc.paddress.address<39:12>;
    else
        PAR.F      = '1';
        PAR.FST    = AArch32.PARFaultStatusSD(addrdesc.fault);
        PAR.LPAE   = '0';
        PAR<31:16> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";       // IMPDEF
    return;
```

## Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

```
// AArch32.PARFaultStatusLD()
// =====
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(32) syndrome;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome<1:0> = if fault.level == 1 then '01' else '10';
        syndrome<5:2> = '1111';
    else
        syndrome = AArch32.FaultStatusLD(TRUE, fault);
    return syndrome<5:0>;
```

## Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```
// AArch32.PARFaultStatusSD()
// =====
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
    bits(32) syndrome;

    syndrome = AArch32.FaultStatusSD(TRUE, fault);
    return syndrome<12,10,3:0>;
```

## Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

```
// AArch32.PARInnerAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;

    if memattrs.memtype == MemType_Device then
        if memattrs.device == DeviceType_nGnRnE then
            result = '001'; // Non-cacheable
        elsif memattrs.device == DeviceType_nGnRE then
            result = '011'; // Non-cacheable
    else
        MemAttrHints inner = memattrs.inner;
        if inner.attrs == MemAttr_NC then
            result = '000'; // Non-cacheable
        elsif inner.attrs == MemAttr_WB && inner.hints<0> == '1' then
            result = '101'; // Write-Back, Write-Allocate
        elsif inner.attrs == MemAttr_WT then
            result = '110'; // Write-Through
        elsif inner.attrs == MemAttr_WB && inner.hints<0> == '0' then
            result = '111'; // Write-Back, no Write-Allocate
    return result;
```

## Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

```
// AArch32.PAROuterAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;

    if memattrs.memtype == MemType_Device then
        result = bits(2) UNKNOWN;
    else
        MemAttrHints outer = memattrs.outer;
        if outer.attrs == MemAttr_NC then
            result = '00'; // Non-cacheable
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elsif outer.attrs == MemAttr_WT && outer.hints<0> == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '0' then
            result = '11'; // Write-Back, no Write-Allocate
    return result;
```



```

// AArch32.DC()
// =====
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    AccType acctype = AccType\_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;
    cache.cachetype = CacheType\_Data;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope == CacheOpScope\_SetWay then
        cache.shareability = Shareability\_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(ZeroExtend(regval), CacheType\_Data);

        if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            ((!ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1') || (ELUsingAArch32(EL2) && HCR.SWIO == '1') ||
            (!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))
            cache.cacheop = CacheOp\_CleanInvalidate;
        CACHE\_OP(cache);
        return;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid = ASID[];
    else
        cache.is_asid_valid = FALSE;

    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp\_Invalidate;
    vaddress = regval;

    size = 0; // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress = ZeroExtend(vaddress);

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch32.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        memattr = memaddrdesc.memattr;
        cache.paddress = memaddrdesc.paddress;
        if opscope == CacheOpScope\_PoC then
            cache.shareability = memattr.shareability;
        else
            cache.shareability = Shareability\_NSH;
    else
        cache.shareability = Shareability\_UNKNOWN;
        cache.paddress = FullAddress\_UNKNOWN;

```

```

if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled()
    && ((!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))
    cache.cacheop = CacheOp\_CleanInvalidate;

CACHE\_OP(cache);
return;

```

## Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```

// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

    boolean match;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        ss = if vaddress<31:5> == CurrentSecurityState();
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && ss == 0 && ! SS\_NonSecureIsSecure then
                match_word<UInt(vaddress<4:2>) + 24> = '1'; // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1'; // Secure vectors (or no EL3)

            if (if HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVBAR<31:5> &&
                ss == 0 && SS\_SecureIsSecure) then
                match_word<UInt(vaddress<4:2>) + 8> = '1'; // Monitor vectors

        // Mask out bits not corresponding to vectors.
        bits(32) mask;
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elseif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHDAPA);

        if !IsZero(vaddress<1:0>) && match then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);
    else
        match = FALSE;

    return match;

```

## Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && 0 && ! NonSecureOnlyImplementationIsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;

```

## Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,
                                           integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumBreakpointsImplemented\(\);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT IN {'0x01'};
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool\(Unpredictable\_BPMATCHHALF\);
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool\(Unpredictable\_BPMISMATCHHALF\);
    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool\(Unpredictable\_BPMATCHHALF\);
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool\(Unpredictable\_BPMISMATCHHALF\);

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```



```

// AArch32.BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.
integer n = n_in;

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplemented() then
    Constraint c;
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR[n].E == '0' then return (FALSE,FALSE);

context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR[n].BT;
Constraint c;

if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
    (dbgtype IN {'010x'} && HaltOnBreakpointOrWatchpoint()) || // Address mismatch
    (! (dbgtype IN {'0x0x'}) && !context_aware) || // Context match
    (dbgtype IN {'1xxx'} && !HaveEL(EL2))) then // EL2 extension
    (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype IN {'0x0x'});
mismatch   = (dbgtype IN {'010x'});
match_vmid = (dbgtype IN {'10xx'});
match_cid1 = (dbgtype IN {'xx1x'});
match_cid2 = (dbgtype IN {'11xx'});
linked      = (dbgtype IN {'xx11'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, or if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE,FALSE);

// Do the comparison.
boolean BVR_match;
if match_addr then
    boolean byte_select_match;
    byte = UInt(vaddress<1:0>);
    assert byte IN {0,2}; // "vaddress" is halfword aligned
    byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    integer top = 31;
    BVR_match = (vaddress<top:2> == DBGBCR[n]<top:2>) && byte_select_match;

elseif match_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBCR[n]<31:0>);
boolean BXVR_match;

```

```

if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;
    if ELUsingAArch32\(EL2\) then
        vmid = ZeroExtend(VTTBR.VMID, 16);
        bvr_vmid = ZeroExtend(DBG BXVR[n]<7:0>, 16);
    elseif !Have16bitVMID\(\) || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBG BXVR[n]<7:0>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBG BXVR[n]<15:0>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
            vmid == bvr_vmid);
    elseif match_cid2 then
        BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt\(\) || HaveV82Debug\(\)) &&
            EL2Enabled\(\) &&
            !ELUsingAArch32\(EL2\) &&
            DBG BXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

return (match && !mismatch, !match && mismatch);

```



```

// AArch32.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC_in, bit HMC_in,
                           bits(2) PxC_in, boolean linked_in, bits(4) LBN,
                           boolean isbreakpt, boolean ispriv)

// "SSC_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
bits(2) SSC = SSC_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpt);
if c == Constraint_DISABLED then return FALSE;
// Otherwise the HMC,SSC,PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.

PL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpt && HMC == '0' && PxC == '00' && SSC != '11';

boolean priv_match;
if !ispriv && !isbreakpt then
    priv_match = PL0_match;
elsif SSU_match then
    priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
else
    case PSTATE.EL of
        when EL3 priv_match = PL1_match; // EL3 and EL1 are both PL1
        when EL2 priv_match = PL2_match;
        when EL1 priv_match = PL1_match;
        when EL0 priv_match = PL0_match;

boolean security_state_match;
ss = CurrentSecurityState();
case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = ss == SS_NonSecure; // Non-secure only
    when '10' security_state_match = ss == SS_Secure; // Secure only
    when '11' security_state_match = (HMC == '1' || ss == SS_Secure); // HMC=1 -> Both, 0 -> Secure

integer lbn;
if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
    last_ctx_cmp = NumBreakpointsImplemented() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE; // Disabled
            when Constraint_NONE linked = FALSE; // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

boolean linked_match;
if linked then
    vaddress = bits(32) UNKNOWN;
    linked_to = TRUE;
    (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

```

```
return priv_match && security_state_match && (!linked || linked_match);
```

## Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    ss = return CurrentSecurityStateAArch32.GenerateDebugExceptionsFrom();
    return(PSTATE.EL, AArch32.GenerateDebugExceptionsFromIsSecure(PSTATE.EL, ss);());
```

## Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, AArch32.GenerateDebugExceptionsFrom(bits(2) from_state,
secure)
    if ! SecurityState from_state)
        if !ELUsingAArch32(DebugTargetFrom(from_state)) then
(secure)) then
            mask = '0'; // No PSTATE.D in AArch32 state
            return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);
(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        assert from_el != ) && secure then
assert from != EL2; // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from_el == if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from_el != enabled = from != EL2;

    return enabled;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch32.CheckForPMUOverflow()
    if !ELUsingAArch32(EL1) then
        AArch64.CheckForPMUOverflow();
        return;
    bit hpme;
    if HaveEL(EL2) then
        hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    boolean pmuirq;
    bit E;
    pmuirq = PMCR.E == '1' && PMINTENSET.C == '1' && PMOVSSET.C == '1';
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch32.PMUCounterIsHyp(idx) then hpme else PMCR.E;
            if E == '1' && PMINTENSET<idx> == '1' && PMOVSSET<idx> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR.)
```

## Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```
// AArch32.ClearEventCounters()
// =====
// Zero all the event counters.

AArch32.ClearEventCounters()
    if HaveAArch64() then
        // Force the counter to be cleared as a 64-bit counter.
        AArch64.ClearEventCounters();
        return;

    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR[idx] = Zeros();
```



```

// AArch32.CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.
// Return TRUE if counter "idx" should count its event. For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch32.CountPMUEvents(integer idx)
    assert idx == CYCLE_COUNTER_ID || idx < GetNumEventCounters();
    if !ELUsingAArch32(EL1) then return AArch64.CountPMUEvents(idx);
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch32.PMUCounterIsHyp(idx);
    ss =
    // Main enable controls
    if idx == CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
        enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
    else
        if resvd_for_el2 then
            E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
        else
            E = PMCR.E;
        enabled = E == '1' && PMCNTENSET<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMUv3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Not executing at EL0, or SDER.SUNIDEN == 0
    if HaveEL(EL3) && ss == && SS_SecureIsSecure then
    () then
        spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
        if !ELUsingAArch32(EL3) && HavePMUv3p7() then
            prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
        else
            prohibited = spme == '0';
        if prohibited && PSTATE.EL == EL0 then
            prohibited = SDER.SUNIDEN == '0';

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = HDCR.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen

```

```

frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() then
    bits(5) hpmn;
    if HaveEL(EL2) then
        hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        ovflws = ZeroExtend(PMOVSSET<GetNumEventCounters()-1:0>);
        bit FZ;
        if resvd_for_el2 then
            FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
            ovflws<UInt(hpmn)-1:0> = Zeros();
        else
            FZ = PMCR.FZO;
            if HaveEL(EL2) && UInt(hpmn) < GetNumEventCounters() then
                ovflws<GetNumEventCounters()-1:UInt(hpmn)> = Zeros();
            frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    enabled = enabled && (PMCR.DP == '0');
    // Otherwise whether event counting is prohibited does not affect the cycle counter
    prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR.DP.
if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && ss == SS_Secure && SS_SecureIsSecure then
        sccd = if ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
        if sccd == '1' then prohibited = TRUE;
    if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR else PMEVTYPER[idx];

P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
    when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';

return !debug && enabled && !prohibited && !filtered && !frozen;

```

## Library pseudocode for aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

```
// AArch32.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch32.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters\(\);
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled\(\) then
        n = UInt(if !ELUsingAArch32\(EL2\) then MDCR_EL2.HPMN else HDCR.HPMN);
        if n > total_counters || (!HaveFeatHPMN0\(\) && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                                                    Unpredictable\_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

```
// AArch32.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
    if HaveAArch64\(\) then
        // Force the counter to be incremented as a 64-bit counter.
        AArch64.IncrementEventCounter(idx, increment);
        return;

    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    PMEVCNTR[idx] = new_value<31:0>;
    ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        PMOVSR<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters\(\) then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

## Library pseudocode for aarch32/debug/pmu/AArch32.PMUCounterIsHyp

```
// AArch32.PMUCounterIsHyp
// =====
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch32.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
        bits(5) hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        resvd_for_el2 = n >= UInt(hpmn);
        if UInt(hpmn) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(hpmn)) then
            resvd_for_el2 = boolean UNKNOWN;
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.PMUCycle

```
// AArch32.PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if AArch32.CountPMUEvents(idx) then
                accumulated = PMUEventAccumulator[idx];
                AArch32.IncrementEventCounter(idx, accumulated);
                PMUEventAccumulator[idx] = 0;

    integer old_value;
    integer new_value;
    integer ovflw;
    if (AArch32.CountPMUEvents(CYCLE_COUNTER_ID) &&
        (PMCR.LC == '1' || PMCR.D == '0' || HasElapsed64Cycles())) then
        old_value = UInt(PMCCNTR);
        new_value = old_value + 1;
        PMCCNTR = new_value<63:0>;

        ovflw = if PMCR.LC == '1' then 64 else 32;

        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET.C = '1';
            PMOVSR.C = '1';

    AArch32.CheckForPMUOverflow();
```

## Library pseudocode for aarch32/debug/pmu/AArch32.PMUSwIncrement

```
// AArch32.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && ! CurrentSecurityStateIsSecure() == SS_NonSecure &&() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityStateIsSecure() == SS_Secure;
    ();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

integer top = 31;
bottom = if DBGWVR[n]<2> == '1' then 2 else 3;          // Word or doubleword
byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
// DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
else
    LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1)); MSB = (DBGWCR[n].BAS + LSB);
    if !IsZero(MSB AND (MSB - 1)) then                // Not contiguous
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
        bottom = 3;                                    // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
    Constraint c;
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
    assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    case c of
        when Constraint_DISABLED return FALSE;        // Disabled
        when Constraint_NONE     mask = 0;            // No masking
        // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

boolean WVR_match;
if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
            top = 63;
    WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
        WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
else
    WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

return WVR_match && byte_select_match;
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    // "ispriv" is:
    // * FALSE for all loads, stores, and atomic operations executed at EL0.
    // * FALSE if the access is unprivileged.
    // * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);
    ls_match = FALSE;
    if iswrite then
        ls_match = (DBGWCR[n].LSC<1> != '0');
    else
        ls_match = (DBGWCR[n].LSC<0> != '0');
    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                             (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                             (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;

    return exception;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AlignmentFault(acctype, iswrite, secondstage));
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    long_format = FALSE;
    if route_to_monitor && ! if route_to_monitor && ! CurrentSecurityStateIsSecure() != SS_Secure then
    () then
        long_format = ((TTBCR_S.EAE == '1') ||
            (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                (fault.secondstage && boolean IMPLEMENTATION_DEFINED "Stage 2 synchronous External Abort") ||
                (fault.secondstage && boolean IMPLEMENTATION_DEFINED "Stage 2 synchronous external abort"))))
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    bits(32) syndrome;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
// The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
// Normally, the current translation table format determines the format. For an abort from
// Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
// following applies:
// * The Secure TTBCR.EAE is set to 1.
// * It is taken from Hyp mode.
// * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
long_format = FALSE;
if route_to_monitor && !if route_to_monitor && !CurrentSecurityStateIsSecure\(\) != SS\_Secure then
() then
    long_format = TTBCR.S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
else
    long_format = TTBCR.EAE == '1';

d_side = FALSE;
bits(32) fsr;
if long_format then
    fsr = AArch32.FaultStatusLD(d_side, fault);
else
    fsr = AArch32.FaultStatusSD(d_side, fault);

if route_to_monitor then
    IFSR_S = fsr;
    IFAR_S = vaddress;
else
    IFSR = fsr;
    IFAR = vaddress;

return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault)));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception\_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL\(EL3\) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (HaveRASExt\(\) && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault)));

    ExceptionRecord exception;
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;

    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault\_Alignment then // PC Alignment fault
            exception = ExceptionSyndrome(Exception\_PCAalignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception\_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);
    if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL\(EL3\) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
        (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception\_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32\_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) pe_error_state,
                                     bits(25) full_syndrome)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);

    if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost\(\) && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(full_syndrome);

    route_to_monitor = HaveEL\(EL3\) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
                   (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x10;
    lr_offset = 8;

    bits(2) target_el;
    if route_to_monitor then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_hyp then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, full_syndrome) then
        ClearPendingPhysicalSError\(\);

    fault = AsyncExternalAbort(parity, pe_error_state, extflag);
    vaddress = bits(32) UNKNOWN;

    case target_el of
        when EL3AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2
            exception = AArch32.AbortSyndrome(Exception\_DataAbort, fault, vaddress);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        when EL1AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
    otherwise
        Unreachable\(\);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FM0==1
        assert HCR.TGE == '0' && HCR.FM0 == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FM0 == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IM0==1
        assert HCR.TGE == '0' && HCR.IM0 == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IM0 == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualErrorException

```
// AArch32.TakeVirtualErrorException()
// =====

AArch32.TakeVirtualErrorException(bit extflag, bits(2) pe_error_state, bits(25) full_syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 and AM0==1
        assert HCR.TGE == '0' && HCR.AM0 == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AM0 == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualErrorException(full_syndrome);

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    FaultRecord fault;
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fault = AsyncExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AsyncExternalAbort(FALSE, VESR_EL2.AET, VESR_EL2.ExT);
    else
        fault = AsyncExternalAbort(parity, pe_error_state, extflag);

    ClearPendingVirtualError();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// =====

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH; // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

## Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT      = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
```

## Library pseudocode for aarch32/exceptions/exceptions/ AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if (tid0 == '1' && reg == '0000')                                // FPSID
            || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then    // MVFRx
                if ELUsingAArch32(EL2) then
                    AArch32.SystemAccessTrap(M32_Hyp, 0x8);            // Exception_AdvSIMDFPAccessTrap
                else
                    AArch64.AArch32SystemAccessTrap(EL2, 0x8);          // Exception_AdvSIMDFPAccessTrap
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception_Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap                ec = 0x01;
        when Exception_CP15RRTTrap            ec = 0x03;
        when Exception_CP15RRTTrap            ec = 0x04;
        when Exception_CP14RRTTrap            ec = 0x05;
        when Exception_CP14DTTrap              ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap     ec = 0x07;
        when Exception_FPIDTrap                ec = 0x08;
        when Exception_PACTrap                 ec = 0x09;
        when Exception_CP14RRTTrap             ec = 0x0C;
        when Exception_BranchTarget            ec = 0x0D;
        when Exception_IllegalState            ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall          ec = 0x11;
        when Exception_HypervisorCall          ec = 0x12;
        when Exception_MonitorCall             ec = 0x13;
        when Exception_InstructionAbort        ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAalignment            ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort               ec = 0x24;
        when Exception_NV2DataAbort            ec = 0x25;
        when Exception_FPTrappedException      ec = 0x28;
        otherwise                             Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\)) ||
            (EL2Enabled\(\) && !ELUsingAArch32\(EL2\) && HCR_EL2.TGE == '1'));
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception\_InstructionAbort, Exception\_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif exceptype == Exception\_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0'; // Secure state
    elseif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    else
        PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

## Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// =====

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

## Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64\(\) then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX = '1';
    FPEXC.TFV = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,I0F
    FPEXC<10:8> = '111'; // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// =====
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL\(EL2\);

    if !ELUsingAArch32\(EL2\) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCEXception(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond\(\) != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64\(\) then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCEXception(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCEXception

```
// AArch32.TakeHVCEXception()
// =====

AArch32.TakeHVCEXception(bits(16) immediate)
    assert HaveEL\(EL2\) && ELUsingAArch32\(EL2\);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;

    exception = ExceptionSyndrome\(Exception\_HypervisorCall\);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// =====

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// =====

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// =====
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                    integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && ! CurrentSecurityStateIsSecure() == SS_NonSecure && ELUsingAArch32(EL2);

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                 integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elseif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                        integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityStateIsSecure() == SS_Secure;
    (,);
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTL.R.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEEnabled

```
// AArch32.CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEEnabled(boolean fpxc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!EL2Enabled() || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) && !ELUsingAArch32(EL2)
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPEEnabled();
        AArch64.CheckFPAdvSIMDEnabled();
    elsif PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL2)
        if fpxc_check && HCR_EL2.RW == '0' then
            fpxc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpxc_en == '0' then UNDEFINED;
            AArch64.CheckFPEEnabled();
        else
            cpacr_asedis = CPACR.ASEDIS;
            cpacr_cp10 = CPACR.cp10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) && !CurrentSecurityStateIsSecure() == SS_NonSecure then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
                if NSACR.cp10 == '0' then cpacr_cp10 = '00';

            if PSTATE.EL != EL2 then
                // Check if Advanced SIMD disabled in CPACR
                if advsimd && cpacr_asedis == '1' then UNDEFINED;

                // Check if access disabled in CPACR
                boolean disabled;
                case cpacr_cp10 of
                    when '00' disabled = TRUE;
                    when '01' disabled = PSTATE.EL == EL0;
                    when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                    when '11' disabled = FALSE;
                if disabled then UNDEFINED;

            // If required, check FPEXC enabled bit.
            if fpxc_check && FPEXC.EN == '0' then UNDEFINED;

            AArch32.CheckFPAdvSIMDTrap(advsimd); // Also check against HCPTR and CPTR_EL3
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if (ifHaveEL(EL3) && !ELUsingAArch32(EL3) &&
            CPTR_EL3.TFP == '1' && EL3SDDTrapPriority()) then
            UNDEFINED;

        ss = CurrentSecurityState();
        if HaveEL(EL2) && ss != SS_Secure && !SS_SecureIsSecure then
            ( ) then
                hcptr_tase = HCPTR.TASE;
                hcptr_cp10 = HCPTR.TCP10;

                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    // Check if access disabled in NSACR
                    if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                    if NSACR.cp10 == '0' then hcptr_cp10 = '1';

                    // Check if access disabled in HCPTR
                    if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                        exception = ) && ! IsSecure() then
                            // Check if access disabled in NSACR
                            if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                            if NSACR.cp10 == '0' then hcptr_cp10 = '1';

                            // Check if access disabled in HCPTR
                            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                                exception.syndrome<24:20> = ConditionSyndrome();

                                if advsimd then
                                    exception.syndrome<5> = '1';
                                else
                                    exception.syndrome<5> = '0';
                                    exception.syndrome<3:0> = '1010';          // coproc field, always 0xA

                                if PSTATE.EL == EL2 then
                                    AArch32.TakeUndefInstrException(exception);
                                else
                                    AArch32.TakeHypTrapException(exception);

                                if HaveEL(EL3) && !ELUsingAArch32(EL3) then
                                    // Check if access disabled in CPTR_EL3
                                    if CPTR_EL3.TFP == '1' then
                                        if Halted() && EDSCR.SDD == '1' then
                                            UNDEFINED;
                                        else then
                                            // Check if access disabled in CPTR_EL3
                                            if CPTR_EL3.TFP == '1' then
                                                AArch64.AdvSIMDFPAccessTrap(EL3);

        return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUnDefOrTrap

```
// AArch32.CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUnDefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUnDefOrTrap(Zeros(16));
    else
        route_to_hyp = EL2Enabled() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            exception.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWfxTrap

```
// AArch32.CheckForWfxTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWfxTrap(bits(2) target_el, WfxType wfxtype)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWfxTrap(target_el, wfxtype);
        return;

    boolean is_wfe = wfxtype == WfxType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WfxTrap(wfxtype, target_el);

        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception_WfxTrap);
            exception.syndrome<24:20> = ConditionSyndrome();

            case wfxtype of
                when WfxType_WFI
                    exception.syndrome<0> = '0';
                when WfxType_WFE
                    exception.syndrome<0> = '1';

            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[.].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
    next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
                     '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[.].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// =====
// Trapped system register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype == Exception_Uncategorized then
        return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTTrap, Exception_CP15RTTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;          // CRn, Reg in case of VMRS
        iss<8:5>   = instr<15:12>;          // Rt
        iss<9>     = '0';                   // RES0

        if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>;          // opc2
            iss<16:14> = instr<23:21>;          // opc1
            iss<4:1>   = instr<3:0>;           // CRm
        else //VMRS Access
            iss<19:17> = '000';               //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';               //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';              //CRm - Hardcoded for VMRS
        elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
            // Trapped MRRC/MCRR, VMRS/VMSR
            iss<19:16> = instr<7:4>;          // opc1
            iss<13:10> = instr<19:16>;          // Rt2
            iss<8:5>   = instr<15:12>;          // Rt
            iss<4:1>   = instr<3:0>;           // CRm
        elsif exception.exceptype == Exception_CP14DTTTrap then
            // Trapped LDC/STC
            iss<19:12> = instr<7:0>;          // imm8
            iss<4>     = instr<23>;           // U
            iss<2:1>   = instr<24,21>;        // P,W
            if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
                iss<8:5> = bits(4) UNKNOWN;
                iss<3>   = '1';
            iss<0> = instr<20>;                // Direction

        exception.syndrome<24:20> = ConditionSyndrome();
        exception.syndrome<19:0>  = iss;

    return exception;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && ! CurrentSecurityStateIsSecure() == SS_NonSecure && ELUsingAArch32(EL2)

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// =====

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
    AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC,
                             AccType_AT, AccType_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return fsr;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC,
            AccType\_AT, AccType\_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain; // Domain field (data fault only)

    return fsr;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(25) iss = Zeros();

    if HaveRASExt() && IsAsyncAbort(fault) then
        iss<11:10> = fault.errortype; // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fslwalk &&
            (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType\_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
            iss<24:14> = LSInstructionSyndrome();

        if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT, AccType\_ATPAN} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fslwalk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return iss;
```

## Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault
bits(5) EncodeSDFSC(Fault statuscode, integer level)

bits(5) result;
case statuscode of
  when Fault\_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
  when Fault\_Alignment
    result = '00001';
  when Fault\_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
  when Fault\_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
  when Fault\_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
  when Fault\_SyncExternal
    result = '01000';
  when Fault\_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
  when Fault\_SyncParity
    result = '11001';
  when Fault\_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
  when Fault\_AsyncParity
    result = '11000';
  when Fault\_AsyncExternal
    result = '10110';
  when Fault\_Debug
    result = '00010';
  when Fault\_TLBConflict
    result = '10000';
  when Fault\_Lockdown
    result = '10100'; // IMPLEMENTATION DEFINED
  when Fault\_Exclusive
    result = '10101'; // IMPLEMENTATION DEFINED
  when Fault\_ICacheMaint
    result = '00100';
  otherwise
    Unreachable\(\);

return result;
```

## Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// =====

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm\_C(imm12, PSTATE.C);

return imm32;
```

## Library pseudocode for aarch32/functions/common/A32ExpandImm\_C

```
// A32ExpandImm_C()
// =====

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

## Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

(SRTYPE, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    SRTYPE shift_t;
    integer shift_n;
    case srtype of
        when '00'
            shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
        when '01'
            shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRTYPE_RRX; shift_n = 1;
            else
                shift_t = SRTYPE_ROR; shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

## Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

SRTYPE DecodeRegShift(bits(2) srtype)
    SRTYPE shift_t;
    case srtype of
        when '00' shift_t = SRTYPE_LSL;
        when '01' shift_t = SRTYPE_LSR;
        when '10' shift_t = SRTYPE_ASR;
        when '11' shift_t = SRTYPE_ROR;
    return shift_t;
```

## Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/RRX\_C

```
// RRX_C()
// =====

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

## Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift\_C(value, srtype, amount, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/Shift\_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType\_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType\_LSL
                (result, carry_out) = LSL\_C(value, amount);
            when SRType\_LSR
                (result, carry_out) = LSR\_C(value, amount);
            when SRType\_ASR
                (result, carry_out) = ASR\_C(value, amount);
            when SRType\_ROR
                (result, carry_out) = ROR\_C(value, amount);
            when SRType\_RRX
                (result, carry_out) = RRX\_C(value, carry_in);

    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm\_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

## Library pseudocode for aarch32/functions/common/VBitOps

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

## Library pseudocode for aarch32/functions/common/VCGEType

```
enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

## Library pseudocode for aarch32/functions/common/VCGTtype

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

## Library pseudocode for aarch32/functions/common/VFPNegMul

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

## Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
        (EL2Enabled() && !ELUsingAArch32(EL2))) then
        AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((!(major IN {4,14}) && HSTR<major> == '1') ||
            (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

### Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

### Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

### Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    acctype = AccType\_ATOMIC;
    iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

### Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDorFPEEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDorFPEEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

CheckAdvSIMDOrVFPEEnabled(boolean include_fpexc_check, boolean advsimd)
  AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
  return;
```

## Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

CheckCryptoEnabled32()
  CheckAdvSIMDEnabled();
  // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
  return;
```

## Library pseudocode for aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

CheckVFPEEnabled(boolean include_fpexc_check)
  advsimd = FALSE;
  AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
  return;
```

## Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
  assert N IN {16,32,64};
  rounding = FPRoundingMode(fpcr);
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType\_Infinity); inf2 = (type2 == FPType\_Infinity);
    zero1 = (type1 == FPType\_Zero); zero2 = (type2 == FPType\_Zero);
    if inf1 && inf2 && sign1 == sign2 then
      result = FPDefaultNaN(fpcr);
      FPProcessException(FPExc\_InvalidOp, fpcr);
    elif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
      result = FPInfinity('0');
    elif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
      result = FPInfinity('1');
    elif zero1 && zero2 && sign1 != sign2 then
      result = FPZero(sign1);
    else
      result_value = (value1 - value2) / 2.0;
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero(result_sign);
      else
        result = FPRound(result_value, fpcr);
  return result;
```

## Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPCRTType_Infinity); inf2 = (type2 == FPCRTType_Infinity);
        zero1 = (type1 == FPCRTType_Zero); zero2 = (type2 == FPCRTType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPCRTType_Infinity); inf2 = (type2 == FPCRTType_Infinity);
        zero1 = (type1 == FPCRTType_Zero); zero2 = (type2 == FPCRTType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0');
        result = FPSub(two, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// =====

FPCRTType StandardFPSCRValue()
    bits(32) upper = '00000000000000000000000000000000';
    bits(32) lower = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000000000000000';
    return upper : lower;
```

## Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// =====

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                                boolean iswrite)

    bit A;
    if PSTATE.EL == EL0 && !ELUsingAArch32\(S1TranslationRegime\(\)\) then
        A = SCTLRL.A; //use AArch64 register, when higher Exception level is using AArch64
    elsif PSTATE.EL == EL2 then
        A = HSCTLRL.A;
    else
        A = SCTLRL.A;
    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType\_ATOMIC, AccType\_ATOMICRW, AccType\_ORDEREDATOMIC,
                          AccType\_ORDEREDATOMICRW, AccType\_ATOMICLS64, AccType\_A32LSMD };
    ordered = acctype IN { AccType\_ORDERED, AccType\_ORDEREDRW, AccType\_LIMITEDORDERED,
                          AccType\_ORDEREDATOMIC, AccType\_ORDEREDATOMICRW };
    vector = acctype == AccType\_VEC;

    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```



```

// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned]
    boolean ispair = FALSE;
    return AArch32.MemSingle[address, size, acctype, aligned, ispair];

// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;

```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadData

```
Hint_PreloadData(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// =====

bits(8*size) MemA(bits(32) address, integer size)
    acctype = AccType\_ATOMIC;
    return Mem\_with\_type[address, size, acctype];

// MemA[] - assignment form
// =====

MemA(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType\_ATOMIC;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

### Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// =====

bits(8*size) MemO(bits(32) address, integer size)
    acctype = AccType\_ORDERED;
    return Mem\_with\_type[address, size, acctype];

// MemO[] - assignment form
// =====

MemO(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType\_ORDERED;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

### Library pseudocode for aarch32/functions/memory/MemS

```
// MemS[] - non-assignment form
// =====
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS(bits(32) address, integer size)
    acctype = AccType\_A32LSMD;
    return Mem\_with\_type[address, size, acctype];

// MemS[] - assignment form
// =====
// Memory accessor for streaming store multiple instructions

MemS(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType\_A32LSMD;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// =====

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType\_NORMAL;
    return Mem\_with\_type[address, size, acctype];

// MemU[] - assignment form
// =====

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_NORMAL;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemU\_unpriv

```
// MemU_unpriv[] - non-assignment form
// =====

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType\_UNPRIV;
    return Mem\_with\_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_UNPRIV;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```



```

// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    boolean ispair = FALSE;
    return Mem_with_type[address, size, acctype, ispair];

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then

        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem_with_type[address, size, acctype, ispair] = value_in;

Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory

```

```

// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
c = ConstrainUnpredictable\(Unpredictable\_DEVPAGE2\);
assert c IN {Constraint\_FAULT, Constraint\_NONE};
if c == Constraint\_NONE then aligned = TRUE;

for i = 1 to size-1
    AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
else
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

```

### Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```

// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);
if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
    AArch64.ESBOperation\(\);
    return;

route_to_monitor = HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) && (HCR.TGE == '1' || HCR.AMO == '1');

bits(5) target;
if route_to_monitor then
    target = M32\_Monitor;
elseif route_to_hyp || PSTATE.M == M32\_Hyp then
    target = M32\_Hyp;
else
    target = M32\_Abort;

boolean mask_active;
if CurrentSecurityStateIsSecure\(\) == SS\_Secure then
    mask_active = TRUE;
elseif target == M32\_Monitor then
    mask_active = SCR.AW == '1' && (!HaveEL\(EL2\) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
    mask_active = target == M32\_Abort || PSTATE.M == M32\_Hyp;

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32\(target\);
intdis = Halted\(\) || ExternalDebugInterruptsDisabled\(el\);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending\(\) then
    syndrome32 = AArch32.PhysicalSErrorSyndrome\(\);
    DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    ClearPendingPhysicalSError\(\);

return;

```

### Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();

```

## Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

```
// AArch32.ReportDeferredSError()
// =====
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
    bits(32) target;
    target<31> = '1'; // A
    syndrome = Zeros(16);
    if PSTATE.EL == EL2 then
        syndrome<11:10> = AET; // AET
        syndrome<9> = ExT; // EA
        syndrome<5:0> = '010001'; // DFSC
    else
        syndrome<15:14> = AET; // AET
        syndrome<12> = ExT; // ExT
        syndrome<9> = TTBCR.EAE; // LPAE
        if TTBCR.EAE == '1' then // Long-descriptor format
            syndrome<5:0> = '010001'; // STATUS
        else // Short-descriptor format
            syndrome<10,3:0> = '10110'; // FS
    if HaveAArch64() then
        target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

## Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

## Library pseudocode for aarch32/functions/ras/AArch32.vESB0Operation

```
// AArch32.vESB0Operation()
// =====
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESB0Operation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESB0Operation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        HCR.VA = '0'; // Clear pending virtual SError

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// =====

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;    // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

## Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// =====

ALUExceptionReturn(bits(32) address)
  if PSTATE.EL == EL2 then
    UNDEFINED;
  elsif PSTATE.M IN {M32\_User,M32\_System} then
    Constraint c = ConstrainUnpredictable(Unpredictable\_ALUEXCEPTIONRETURN);
    assert c IN {Constraint\_UNDEF, Constraint\_NOP};
    case c of
      when Constraint\_UNDEF
        UNDEFINED;
      when Constraint\_NOPEndOfInstruction();
    else
      AArch32.ExceptionReturn(address, SPSR[]);
```

## Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
  if CurrentInstrSet() == InstrSet\_A32 then
    BXWritePC(address, BranchType\_INDIR);
  else
    BranchWritePC(address, BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// =====

BXWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = address_in;
  if address<0> == '1' then
    SelectInstrSet(InstrSet\_T32);
    address<0> = '0';
  else
    SelectInstrSet(InstrSet\_A32);
    // For branches to an unaligned PC counter in A32 state, the processor takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable\_A32FORCEALIGNPC) then
      address<1> = '0';
    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(address, branch_type, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// =====

BranchWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = address_in;
  if CurrentInstrSet() == InstrSet\_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
  boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
  BranchTo(address, branch_type, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// =====
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
    bits(32) address = address_in;
    assert CurrentInstrSet\(\) == InstrSet\_T32;
    address<0> = '0';
    boolean branch_conditional = TRUE;
    BranchTo(address, BranchType\_DIR, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    return vreg<base+63:base>;

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    vreg<base+63:base> = value;
    V[n DIV 2] = vreg;
    return;
```

## Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// =====

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return \_Dclone[n];
```

## Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// =====

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// =====

bits(32) LR
    return R[14];
```

## Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of // Select index by mode:      usr fiq irq svc abt und hyp
        when 8      result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9      result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10     result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11     result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12     result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13     result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14     result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise   result = n;

    return result;
```

## Library pseudocode for aarch32/functions/registers/Monitor\_mode\_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

## Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// =====

bits(32) PC
    return R[15]; // This includes the offset from AArch32 state
```

## Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

## Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n] = value;
    return;
```

## Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// =====

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

## Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =====

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet\_A32 then 8 else 4);
        return \_PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

## Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    integer result;
    case mode of
        when M32\_User      result = usr;    // User mode
        when M32\_FIQ      result = fiq;    // FIQ mode
        when M32\_IRQ      result = irq;    // IRQ mode
        when M32\_Svc      result = svc;    // Supervisor mode
        when M32\_Abort     result = abt;    // Abort mode
        when M32\_Hyp       result = hyp;    // Hyp mode
        when M32\_Undef     result = und;    // Undefined mode
        when M32\_System    result = usr;    // System mode uses User mode registers
        otherwise         Unreachable(); // Monitor mode

    return result;
```

## Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if if ! CurrentSecurityStateIsSecure() != () then assert mode != SS_Secure then assert mode != M32_Monitor
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookupRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if if ! CurrentSecurityStateIsSecure() != SS_Secure then assert mode != () then assert mode != M32_Monitor
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookupRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookupRIndex(n, mode)]<31:0> = value;

    return;
```

## Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =====

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    return vreg<base+31:base>;

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    vreg<base+31:base> = value;
    V[n DIV 4] = vreg;
    return;
```

## Library pseudocode for aarch32/functions/registers/SP

```
// SP - assignment form
// =====

SP = bits(32) value
  R[13] = value;
  return;

// SP - non-assignment form
// =====

bits(32) SP
  return R[13];
```

## Library pseudocode for aarch32/functions/registers/\_Dclone

```
array bits(64) _Dclone[0..31];
```

## Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
  bits(32) new_pc = new_pc_in;
  SynchronizeContext\(\);
  // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
  // mechanism
  SetPSTATEFromPSR\(spsr\);
  ClearExclusiveLocal\(ProcessorID\(\)\);
  SendEventLocal\(\);

  if PSTATE.IL == '1' then
    // If the exception return is illegal, PC[1:0] are UNKNOWN
    new_pc<1:0> = bits(2) UNKNOWN;
  else
    // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
    if PSTATE.T == '1' then
      new_pc<0> = '0'; // T32
    else
      new_pc<1:0> = '00'; // A32

  boolean branch_conditional = !(AArch32.CurrentCond\(\) IN {'111x'});
  BranchTo\(new\_pc, BranchType\_ERET, branch\_conditional\);

  CheckExceptionCatch\(FALSE\); // Check for debug event on exception return
```

## Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// =====

boolean AArch32.ExecutingCP10or11Instr()
  instr = ThisInstr\(\);
  instr_set = CurrentInstrSet\(\);
  assert instr_set IN {InstrSet\_A32, InstrSet\_T32};

  if instr_set == InstrSet\_A32 then
    return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'101x'});
  else // InstrSet_T32
    return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'101x'});
```

### Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// =====

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].
AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].
AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn = UInt(instr<19:16>);
    CRm = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// Read the contents of R[t] and write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```
// Read a value from a virtual address and write it to an AArch32 System register.
AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M    = mode;
    PSTATE.EL    = el;
    PSTATE.nRW   = '1';
    PSTATE.SP    = (if mode IN {M32\_User,M32\_System} then '0' else '1');
    return;
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32\_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

## Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
boolean valid;
case mode of
    when M32_Monitor
        valid = HaveAArch32EL(EL3);
    when M32_Hyp
        valid = HaveAArch32EL(EL2);
    when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
        // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
        // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
        // AArch64, then these modes are EL1 modes.
        // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
        valid = HaveAArch32EL(EL1);
    when M32_User
        valid = HaveAArch32EL(EL0);
    otherwise
        valid = FALSE;          // Passed an illegal mode value
return !valid;
```

## Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100' // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101' // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110' // LR_usr
            if mode IN {M32_Hyp, M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110' // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x' // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x' // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x' // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x' // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x' // LR_mon, SP_mon
            if (!if !HaveEL(EL3) ||) || !CurrentSecurityStateIsSecure() != SS_Secure ||
                mode ==() || mode == M32_Monitor) then UNPREDICTABLE;
    then UNPREDICTABLE;
        when '11110' // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor, M32_Hyp}) then UNPREDICTABLE;
        when '11111' // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

return;
```

## Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;          // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
    return;
```

## Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

## Library pseudocode for aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```

## Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// =====

boolean InITBlock()
    if CurrentInstrSet() == InstrSet\_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

## Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// =====

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

## Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    bits(32) new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr; // UNPREDICTABLE if User or System mode

    return;
```

## Library pseudocode for aarch32/functions/system/SPSRAccessValid

```
// SPSRAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRAccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110' // SPSR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000' // SPSR_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '10010' // SPSR_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100' // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110' // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100' // SPSR_mon
            if (!HaveEL(EL3) || mode == M32_Monitor || !CurrentSecurityStateIsSecure() != SS_Secure) then UNPREDICTABLE;
        () then UNPREDICTABLE;
        when '11110' // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
            otherwise
                UNPREDICTABLE;

    return;
```

### Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// =====

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet\_A32, InstrSet\_T32};
    assert iset IN {InstrSet\_A32, InstrSet\_T32};

    PSTATE.T = if iset == InstrSet\_A32 then '0' else '1';

    return;
```

### Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

### Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSat0(i, N);
    return result;
```

### Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSat0(i, N);
    return result;
```



```

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval);
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID[];
        else
            cache.is_asid_valid = FALSE;

        need_translate = ICInstNeedsTranslation(opscope);

        cache.shareability = Shareability_NSH;
        cache.vaddress = ZeroExtend(regval);
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        wasaligned = TRUE;
        iswrite = FALSE;
        size = 0;
        memaddrdesc = AArch32.TranslateAddress(regval, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;

```

## Library pseudocode for aarch32/predictionrestrict/RestrictPrediction

```
// RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el      = val<25:24>;

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if UInt(target_el) > UInt(PSTATE.EL) then return;

    bit ns = val<26>;
    ss = TargetSecurityState(ns);

    c.security = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

            elsif target_el IN {EL0, EL1} then
                c.is_vmid_valid = TRUE;
                c.all_vmid      = val<27> == '1';
                c.vmid          = ZeroExtend(val<23:16>, 16);           // Only valid if val<27> == '0';
            else
                c.is_vmid_valid = FALSE;
        else
            c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<8> == '1';
        c.asid          = ZeroExtend(val<7:0>, 16);                 // Only valid if val<8> == '0';
    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```



```

// AArch32.DefaultTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit B_in, bit S)
    MemoryAttributes memattrs;
    bits(3) TEX = TEX_in;
    bit C = C_in;
    bit B = B_in;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits\(Unpredictable\_RESTEXCB\);
        TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.memtype = MemType\_Device;
            memattrs.device = DeviceType\_nGnRnE;
            memattrs.shareability = Shareability\_OSH;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.memtype = MemType\_Device;
            memattrs.device = DeviceType\_nGnRE;
            memattrs.shareability = Shareability\_OSH;
        when '00010'
            // Write-through Read allocate
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_WT;
            memattrs.inner.hints = MemHint\_RA;
            memattrs.outer.attrs = MemAttr\_WT;
            memattrs.outer.hints = MemHint\_RA;
            memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
        when '00011'
            // Write-back Read allocate
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_WB;
            memattrs.inner.hints = MemHint\_RA;
            memattrs.outer.attrs = MemAttr\_WB;
            memattrs.outer.hints = MemHint\_RA;
            memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
        when '00100'
            // Non-cacheable
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_NC;
            memattrs.outer.attrs = MemAttr\_NC;
            memattrs.shareability = Shareability\_OSH;
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_WB;
            memattrs.inner.hints = MemHint\_RWA;
            memattrs.outer.attrs = MemAttr\_WB;
            memattrs.outer.hints = MemHint\_RWA;
            memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.memtype = MemType\_Normal;
            memattrs.inner = DecodeSDFAttr(C:B);
            memattrs.outer = DecodeSDFAttr(TEX<1:0>);

            if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
                memattrs.shareability = Shareability\_OSH;
            else

```

```

        memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
    otherwise
        // Reserved, handled above
        Unreachable\(\);

// The Transient hint is not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;
memattrs.tagged          = FALSE;

if memattrs.inner.attrs == MemAttr\_WB && memattrs.outer.attrs == MemAttr\_WB then
    memattrs.xs = '0';
else
    memattrs.xs = '1';

return memattrs;

```

## Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit C, bit B, bit S)

MemoryAttributes memattrs;
PRRR_Type prrr;
NMRR_Type nmrr;

region = UInt(TEX<0>:C:B);          // TEX<2:1> are ignored in this mapping scheme
if region == 6 then
    return MemoryAttributes IMPLEMENTATION_DEFINED;

if regime == Regime\_EL30 then
    prrr = PRRR_S;
    nmrr = NMRR_S;
elseif HaveAArch32EL(EL3) then
    prrr = PRRR_NS;
    nmrr = NMRR_NS;
else
    prrr = PRRR;
    nmrr = NMRR;

base = 2 * region;
attrfield = prrr<base+1:base>;

if attrfield == '11' then          // Reserved, maps to allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESPRRR);

case attrfield of
    when '00'                      // Device-nGnRnE
        memattrs.memtype          = MemType\_Device;
        memattrs.device           = DeviceType\_nGnRnE;
        memattrs.shareability     = Shareability\_OSH;
    when '01'                      // Device-nGnRE
        memattrs.memtype          = MemType\_Device;
        memattrs.device           = DeviceType\_nGnRE;
        memattrs.shareability     = Shareability\_OSH;
    when '10'
        NSn = if S == '0' then prrr.NS0 else prrr.NS1;
        NOSm = prrr<region+24> AND NSn;
        IRn = nmrr<base+1:base>;
        ORn = nmrr<base+17:base+16>;

        memattrs.memtype = MemType\_Normal;
        memattrs.inner   = DecodeSDFAttr(IRn);
        memattrs.outer   = DecodeSDFAttr(ORn);
        if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
            memattrs.shareability = Shareability\_OSH;
        else
            bits(2) sh = NSn:NOSm;
            memattrs.shareability = DecodeShareability(sh);
    when '11'
        Unreachable();

// The Transient hint is not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;
memattrs.tagged          = FALSE;

if memattrs.inner.attrs == MemAttr\_WB && memattrs.outer.attrs == MemAttr\_WB then
    memattrs.xs = '0';
else
    memattrs.xs = '1';

return memattrs;
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elseif (match || mismatch) then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = NoFault();

    boolean d_side = (d_side = (acctype != IsDataAccess(acctype) || acctype == AccType_DC);
    boolean i_side = (acctype == AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        elseif i_side then
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
    && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMATCH(vaddress);
    if size == 4 && !match && AArch32.VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCATCHHALF);

    if match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    if acctype == if acctype IN { AccType_DC, AccType_TTW } then
        if !iswrite then
            return, AccType_IC, AccType_AT, AccType_ATPAN } then
            return NoFault();
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return if acctype == AccType_DC then
                if !iswrite then
                    return NoFault();
            elsif !IsDataAccess(acctype) then
                return();
            elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
                return NoFault();

    match = FALSE;
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;

    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress);
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

## Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)
    FaultRecord fault;

    fault.statuscode = Fault_Debug;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.debugmoe    = debugmoe;
    fault.secondstage = FALSE;
    fault.s2fslwalk   = FALSE;

    return fault;
```

## Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =====
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

## Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

```
// AArch32.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccType acctype, boolean aligned,
                                     bit ntlsmid, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    if acctype == AccType_A32LSMD && ntlsmid == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;

    return !aligned || acctype == AccType_DCZVA;
```



```

// AArch32.S1LDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, SecurityState ss, S1TTWParams walkparams,
                                         Permissions perms, MemType memtype, PASpace paspace,
                                         boolean ispriv, AccType acctype, boolean iswrite)

bit r;
bit w;
bit x;
bit pr;
bit pw;
bit ur;
bit uw;
bit xn;
if HasUnprivileged(regime) then
    // Apply leaf permissions
    case perms.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // R0 at PL1 only
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL

    // Apply hierarchical permissions
    case perms.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr, '0', '0','0'); // Read-only, privileged access

    xn = perms.xn OR perms.xn_table;
    pxn = perms.pxn OR perms.pxn_table;

    ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
    px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

    pan_access = !(acctype IN {AccType\_DC, AccType\_IFETCH, AccType\_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if ss == SS\_Secure && paspace == PAS\_NonSecure then
        x = x AND NOT(walkparams.sif);
else
    // Apply leaf permissions
    case perms.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only

    // Apply hierarchical permissions
    case perms.ap_table<1> of
        when '0' (r,w) = ( r , w ); // No effect
        when '1' (r,w) = ( r , '0'); // Read-only

    xn = perms.xn OR perms.xn_table;
    x = NOT(xn OR (w AND walkparams.wxn));

if acctype == AccType\_IFETCH then
    constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
    if constraint == Constraint\_FAULT && memtype == MemType\_Device then
        return TRUE;
    else
        return x == '0';
elseif acctype IN {AccType\_IC, AccType\_DC} then
    return FALSE;

```

```
elseif iswrite then
    return w == '0';
else
    return r == '0';
```



```

// AArch32.S1SDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, SecurityState ss, Permissions perms_in,
                                         MemType memtype, PASpace paspace, boolean ispriv,
                                         AccType acctype, boolean iswrite)

    Permissions perms = perms_in;
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    SCTLR_Type sctlr;
    if regime == Regime\_EL30 then
        sctlr = SCTLR_S;
    elsif HaveAArch32EL\(EL3\) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

        case perms.ap of
            when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, R0 at PL0
            when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            // '100' is reserved
            when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // R0 at PL1 only
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL (deprecated)
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL
        else // Simplified access permissions model
            case perms.ap<2:1> of
                when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
                when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
                when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // R0 at PL1 only
                when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    pan_access = !(acctype IN {AccType\_DC, AccType\_IFETCH, AccType\_AT});
    if HavePANExt\(\) && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if ss == SS\_Secure && paspace == PAS\_NonSecure then
        x = x AND NOT(if ELUsingAArch32\(EL3\) then SCR.SIF else SCR_EL3.SIF);

    if acctype == AccType\_IFETCH then
        constraint = ConstrainUnpredictable\(Unpredictable\_INSTRDEVICE\);
        if constraint == Constrain\_FAULT && memtype == MemType\_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType\_IC, AccType\_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';

```

## Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

```
// AArch32.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccType acctype, boolean aligned,
                                   MemoryAttributes memattrs)
    if acctype == AccType\_IFETCH || memattrs.memtype != MemType\_Device then
        return FALSE;

    return !aligned || acctype == AccType\_DCZVA;
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(boolean s2fslwalk, S2TTWParams walkparams,
                                       Permissions perms, MemType memtype,
                                       boolean ispriv, AccType acctype,
                                       boolean iswrite)

    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xnx of
            when '00' (px, ux) = ( r , r );
            when '01' (px, ux) = ( '0', r );
            when '10' (px, ux) = ( '0', '0' );
            when '11' (px, ux) = ( r , '0' );

        x = if ispriv then px else ux;
    else
        x = r AND NOT(perms.s2xn);

    if s2fslwalk && walkparams.ptw == '1' && memtype == MemType\_Device then
        return TRUE;
    elseif acctype == AccType\_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
        if constraint == Constraint\_FAULT && memtype == MemType\_Device then
            return TRUE;
        else
            return x == '0';
    elseif acctype IN {AccType\_IC, AccType\_DC} then
        return FALSE;
    elseif iswrite then
        return w == '0';
    else
        return r == '0';
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch32.S2StartLevel(walkparams.sl0);
    levels     = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

## Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// =====
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime\_EL2 then
        // Input Address size
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && IsZero(va<31:iasize>);
    elseif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return IsZero(va<31:lo_iasize>) && IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

```
// AArch32.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
    TLBContext tlbcontext;

    case regime of
        when Regime\_EL2 tlbcontext = AArch32.TLBContextEL2(va);
        when Regime\_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime\_EL30 tlbcontext = AArch32.TLBContextEL30(va);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS2TLBContext

```
// AArch32.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address);
    tlbcontext.cnp         = if HaveCommonNotPrivateTransExt() then VTTBR.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL10

```
// AArch32.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR\_Type contextidr;

    if HaveAArch32EL(EL3) then
        ttbcr      = TTBCR_NS;
        ttbr0      = TTBR0_NS;
        ttbr1      = TTBR1_NS;
        contextidr = CONTEXTIDR_NS;
    else
        ttbcr      = TTBCR;
        ttbr0      = TTBR0;
        ttbr1      = TTBR1;
        contextidr = CONTEXTIDR;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL10;

    if AArch32.EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend(VTTBR.VMID);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID);
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID);

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend(va);

    if HaveCommonNotPrivateTransExt() && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange\_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL2

```
// AArch32.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS\_NonSecure;
    tlbcontext.regime = Regime\_EL2;
    tlbcontext.ia      = ZeroExtend(va);
    tlbcontext.tg      = TGx\_4KB;
    tlbcontext.cnp      = if HaveCommonNotPrivateTransExt() then HTTBR.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL30

```
// AArch32.TLBContextEL30()
// =====
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS_Secure;
    tlbcontext.regime = Regime_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va);

    if HaveCommonNotPrivateTransExt() && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VRange_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/translation/AArch32.AccessUsesEL

```
// AArch32.AccessUsesEL()
// =====
// Determine the privilege associated with the access

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

## Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```
// AArch32.EL2Enabled()
// =====
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS_Secure then
        if !(HaveEL(EL2) && HaveSecureEL2Ext()) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```

## Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =====
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccType acctype,
                                       boolean iswrite, boolean aligned)

    // Prepare fault fields in case a fault is detected
    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    regime = TranslationRegime(PSTATE.EL, acctype);
    ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;
    ss = SecurityStateForRegime(regime);

    // First Stage Translation
    AddressDescriptor ipa;
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, ss, va, acctype,
                                             aligned, iswrite, ispriv);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, ss, va, acctype,
                                                aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va);
        s2fslwalk = FALSE;
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, ss, s2fslwalk, acctype,
                                         aligned, iswrite, ispriv);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);
        else
            return pa;
    else
        return ipa;
```

## Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

```
// AArch32.OutputDomain()
// =====
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    index = 2 * UInt(domain);
    if regime == Regime_EL30 then
        Dn = DACR_S<index+1:index>;
    elsif HaveAArch32EL(EL3) then
        Dn = DACR_NS<index+1:index>;
    else
        Dn = DACR<index+1:index>;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR);

    return Dn;
```



```

// AArch32.S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
SecurityState ss, bits(32) va,
AccType acctype, boolean aligned)

FaultRecord fault = fault_in;
// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
bit default_cacheable;
if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
    if ELStateUsingAArch32(EL2, ss == SS_Secure) then
        default_cacheable = HCR.DC;
    else
        default_cacheable = HCR_EL2.DC;
else
    default_cacheable = '0';

if default_cacheable == '1' then
    // Use default cacheable settings
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RWA;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RWA;
    memattrs.shareability = Shareability_NSH;
    if !ELStateUsingAArch32(EL2, ss == SS_Secure) && HaveMTE2Ext() then
        memattrs.tagged = HCR_EL2.DCT == '1';
    else
        memattrs.tagged = FALSE;
elseif acctype == AccType_IFETCH then
    memattrs.memtype = MemType_Normal;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;
    if AArch32.S1ICacheEnabled(regime) then
        memattrs.inner.attrs = MemAttr_WT;
        memattrs.inner.hints = MemHint_RA;
        memattrs.outer.attrs = MemAttr_WT;
        memattrs.outer.hints = MemHint_RA;
    else
        memattrs.inner.attrs = MemAttr_NC;
        memattrs.outer.attrs = MemAttr_NC;
else
    // Treat memory region as Device
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
        when Regime_EL2 ntlsmd = HSCTLR.nTLSMD;
        when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
else
    ntlsmd = '1';

if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, AddressDescriptor UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va);
oa.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

```

```
return (fault, ipa);
```

### Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```
// AArch32.S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    else
        return HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';
```



```

// AArch32.S1TranslateLD()
// =====
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
                                                         SecurityState ss, bits(32) va,
                                                         AccType acctype, boolean aligned,
                                                         boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fslwalk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    return AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

walkparams = AArch32.GetS1TTWParams(regime, va);

if AArch32.VAIsOutOfRange(regime, walkparams, va) then
    fault.level = 1;
    fault.statuscode = Fault_Translation;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkLD(fault, regime, ss, walkparams, va, ispriv);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

if AArch32.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    if AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
                                         walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         walkstate.baseaddress.paspace,
                                         ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
                                         walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         walkstate.baseaddress.paspace,
                                         ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions
        // The permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = TRUE;
    elseif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
                                         walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         walkstate.baseaddress.paspace,
                                         ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;

```

```

else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = Stage0A(ZeroExtend(va), walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

return (fault, ipa);

```



```

// AArch32.S1TranslateSD()
// =====
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
                                                                SecurityState ss, bits(32) va,
                                                                AccType acctype, boolean aligned,
                                                                boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fslwalk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
    return (fault, ipa, SDType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, ss, va, ispriv);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, SDType UNKNOWN);

domain = AArch32.OutputDomain(regime, walkstate.domain);
SetInGuardedPage(FALSE); // AArch32-VMVA does not guard any pages

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
        when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else
        ntlsmd = '1';

if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif !(acctype IN {AccType_IC, AccType_DC}) && domain == Domain_NoAccess then
    fault.statuscode = Fault_Domain;
elseif domain == Domain_Client then
    if IsAtomicRW(acctype) then
        if AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                            walkstate.memattrs.memtype,
                                            walkstate.baseaddress.paspace,
                                            ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            // The permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = FALSE;
        elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                                walkstate.memattrs.memtype,
                                                walkstate.baseaddress.paspace,
                                                ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            // The permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = TRUE;
        elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                                walkstate.memattrs.memtype,
                                                walkstate.baseaddress.paspace,
                                                ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;

if fault.statuscode != Fault_None then
    fault.domain = walkstate.domain;
    return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

MemoryAttributes memattrs;
if ((acctype == AccType_IFETCH &&

```

```

        (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (acctype != AccType_IFETCH &&
            walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = AArch32.SDStage0A(walkstate.baseaddress, va, walkstate.sdftype);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

return (fault, ipa, walkstate.sdftype);

```



```

// AArch32.S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    SecurityState ss, boolean s2fslwalk,
                                                    AccType acctype, boolean aligned,
                                                    boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
assert IsZero(ipa.paddress.address<51:40>);

if !ELStateUsingAArch32(EL2, ss == SS_Secure) then
    slaarch64 = FALSE;
    return AArch64.S2Translate(fault, ipa, slaarch64, ss, s2fslwalk, acctype,
                                aligned, iswrite, ispriv);

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None;
fault.secondstage = TRUE;
fault.s2fslwalk = s2fslwalk;
fault.ipaddress = ipa.paddress;

walkparams = AArch32.GetS2TTWParams();

if walkparams.vm == '0' then
    // Stage 2 is disabled
    return (fault, ipa);

if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
    fault.statuscode = Fault_Translation;
    fault.level = 1;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S2Walk(fault, walkparams, ipa);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

if AArch32.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    assert !s2fslwalk; // AArch32 does not support HW update of TT
    if AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions
        // The permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = TRUE;
    elseif AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;
MemoryAttributes s2_memattrs;
if ((s2fslwalk &&
    walkstate.memattrs.memtype == MemType_Device) ||
    (acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1'))) ||

```

```

    (acctype != AccType\_IFETCH &&
     walkstate.memattrs.memtype == MemType\_Normal && HCR2.CD == '1')) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
// Output Address
oa = Stage0A(ipa_64, walkparams.tgx, walkstate);
pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);

return (fault, pa);

```

### Library pseudocode for aarch32/translation/translation/AArch32.SDStage0A

```

// AArch32.SDStage0A()
// =====
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStage0A(FullAddress baseaddress, bits(32) va, SDType sdftype)
integer tsize;
case sdftype of
    when SDType\_SmallPage      tsize = 12;
    when SDType\_LargePage     tsize = 16;
    when SDType\_Section       tsize = 20;
    when SDType\_Supersection  tsize = 24;

// Output Address
FullAddress oa;
oa.address = baseaddress.address<51:tsize>:va<tsize-1:0>;
oa.paspace = baseaddress.paspace;
return oa;

```

### Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```

// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccType acctype,
                                           boolean iswrite, boolean aligned,
                                           integer size)

regime = TranslationRegime(PSTATE.EL, acctype);
if !RegimeUsingAArch32(regime) then
    return AArch64.TranslateAddress(ZeroExtend(va, 64), acctype, iswrite,
                                   aligned, size);
result = AArch32.FullTranslate(va, acctype, iswrite, aligned);
if !IsFault(result) then
    result.fault = AArch32.CheckDebug(va, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va);

return result;

```

## Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// =====
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Block;
    else
        return DescriptorType_Invalid;
```

## Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// =====
// Determine the type of the short-descriptor

SDType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDType_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDType_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDType_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDType_LargePage;
    elsif level == 2 && descriptor<1:0> IN {'1x'} then
        return SDType_SmallPage;
    else
        return SDType_Invalid;
```

## Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```
// AArch32.S1IASize()
// =====
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```



```

// AArch32.S1WalkLD()
// =====
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime, SecurityState ss,
                                           S1TTWParams walkparams, bits(32) va, boolean ispriv)

    FaultRecord fault = fault_in;
    bits(3) txsz;
    bits(64) ttbr;
    bit epd;
    if regime == Regime_EL2 then
        ttbr = HTTBR;
        txsz = walkparams.t0sz;
    else
        varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        bits(64) ttbr0;
        bits(64) ttbr1;
        TTBCR_Type ttbcr;
        if regime == Regime_EL30 then
            ttbcr = TTBCR_S;
            ttbr0 = TTBR0_S;
            ttbr1 = TTBR1_S;
        elseif HaveAArch32EL(EL3) then
            ttbcr = TTBCR_NS;
            ttbr0 = TTBR0_NS;
            ttbr1 = TTBR1_NS;
        else
            ttbcr = TTBCR;
            ttbr0 = TTBR0;
            ttbr1 = TTBR1;

        assert ttbcr.EAE == '1';
        if varange == VARange_LOWER then
            txsz = walkparams.t0sz;
            ttbr = ttbr0;
            epd = ttbcr.EPD0;
        else
            txsz = walkparams.t1sz;
            ttbr = ttbr1;
            epd = ttbcr.EPD1;

    if regime != Regime_EL2 && epd == '1' then
        fault.level = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S1IASize(txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
levels = FINAL_LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb));

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global

```

```

walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table = '00';
walkstate.permissions.xn_table = '0';
walkstate.permissions.pxn_table = '0';

indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

integer indexlsb;
DescriptorType descctype;
repeat
    fault.level = walkstate.level;
    indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
    bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000');

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime\_EL10 && AArch32.EL2Enabled(ss) then
        s2fslwalk = TRUE;
        s2acctype = AccType\_TTW;
        s2aligned = TRUE;
        s2write = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fslwalk,
                                                    s2acctype, s2aligned, s2write, ispriv);

        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault\_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.tee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.tee, walkaddress, fault);

    if fault.statuscode != Fault\_None then
        return (fault, TTWState UNKNOWN);

    descctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case descctype of
        when DescriptorType\_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault\_AddressSize;
                return (fault, TTWState UNKNOWN);

                walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12));
                if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<63> == '1' then
                    walkstate.baseaddress.paspace = PAS\_NonSecure;

                if walkparams.hpd == '0' then

```

```

        walkstate.permissions.xn_table = (walkstate.permissions.xn_table OR
                                          descriptor<60>);
        walkstate.permissions.ap_table = (walkstate.permissions.ap_table OR
                                          descriptor<62:61>);
        walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                          descriptor<59>);

        walkstate.level = walkstate.level + 1;
        indexmsb = indexlsb - 1;

    when DescriptorType\_Invalid
        fault.statuscode = Fault\_Translation;
        return (fault, TTWState\_UNKNOWN);

    when DescriptorType\_Page, DescriptorType\_Block
        walkstate.istable = FALSE;

until desctype IN {DescriptorType\_Page, DescriptorType\_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault\_AddressSize;
    return (fault, TTWState\_UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault\_AccessFlag;
    return (fault, TTWState\_UNKNOWN);

walkstate.permissions.xn = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap = descriptor<7:6>: '1';
walkstate.contiguous = descriptor<52>;
if regime == Regime\_EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elsif ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor<11>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>: Zeros(indexlsb));
if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS\_NonSecure;

memattr = descriptor<4:2>;
sh = descriptor<9:8>;
attr = MAIRAttr(UInt(memattr), walkparams.mair);
slaarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64);

return (fault, walkstate);

```



```

// AArch32.S1WalkSD()
// =====
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime, SecurityState ss,
                                         bits(32) va, boolean ispriv)

    FaultRecord fault = fault_in;
    SCTLr_Type sctlr;
    TTBCr_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    // Determine correct translation control registers to use.
    if regime == Regime\_EL30 then
        sctlr = SCTLr_S;
        ttbcr = TTBCr_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elsif HaveAArch32EL\(EL3\) then
        sctlr = SCTLr_NS;
        ttbcr = TTBCr_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        sctlr = SCTLr;
        ttbcr = TTBCr;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    assert ttbcr.EAE == '0';
    ee = sctlr.EE;
    afe = sctlr.AFE;
    tre = sctlr.TRE;
    n = UInt(ttbcr.N);
    bits(32) ttb;
    bits(1) pd;
    bits(2) irgn;
    bits(2) rgn;
    bits(1) s;
    bits(1) nos;
    if n == 0 || IsZero(va<31:(32-n)>) then
        ttb = ttbr0.TTB0:Zeros(7);
        pd = ttbcr.PD0;
        irgn = ttbr0.IRGN;
        rgn = ttbr0.RGN;
        s = ttbr0.S;
        nos = ttbr0.NOS;
    else
        n = 0; // TTBR1 translation always treats N as 0
        ttb = ttbr1.TTB1:Zeros(7);
        pd = ttbcr.PD1;
        irgn = ttbr1.IRGN;
        rgn = ttbr1.RGN;
        s = ttbr1.S;
        nos = ttbr1.NOS;

    // Check if Translation table walk disabled for translations with this Base register.
    if pd == '1' then
        fault.level = 1;
        fault.statuscode = Fault\_Translation;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baseaddress.paspace = if ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
    baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n));

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global. Translations in Short-Descriptor Format

```

```

// always support global & non-global translations.
walkstate.nG          = '1';
walkstate.memattrs    = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level       = 1;
walkstate.istable     = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
    fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00');
    else
        index = ZeroExtend(va<19:12>:'00');

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    if regime == Regime\_EL10 && AArch32.EL2Enabled(ss) then
        s2fslwalk = TRUE;
        s2acctype = AccType\_TTW;
        s2aligned = TRUE;
        s2write = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fslwalk,
                                                    s2acctype, s2aligned, s2write, ispriv);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, fault);

    if fault.statuscode != Fault\_None then
        return (fault, TTWState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDType\_Invalid
            fault.domain = domain;

```

```

    fault.statuscode = Fault\_Translation;
    return (fault, ITWState UNKNOWN);

when SDFType\_Table
    domain = descriptor<8:5>;
    ns     = descriptor<3>;
    pxn    = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10));
    walkstate.level = 2;

when SDFType\_SmallPage
    nG = descriptor<11>;
    s  = descriptor<10>;
    ap = descriptor<9,5:4>;
    tex = descriptor<8:6>;
    c   = descriptor<3>;
    b   = descriptor<2>;
    xn  = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12));
    walkstate.istable = FALSE;

when SDFType\_LargePage
    xn = descriptor<15>;
    tex = descriptor<14:12>;
    nG = descriptor<11>;
    s  = descriptor<10>;
    ap = descriptor<9,5:4>;
    c   = descriptor<3>;
    b   = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16));
    walkstate.istable = FALSE;

when SDFType\_Section
    ns     = descriptor<19>;
    nG     = descriptor<17>;
    s      = descriptor<16>;
    ap     = descriptor<15,11:10>;
    tex    = descriptor<14:12>;
    domain = descriptor<8:5>;
    xn     = descriptor<4>;
    c      = descriptor<3>;
    b      = descriptor<2>;
    pxn    = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20));
    walkstate.istable = FALSE;

when SDFType\_Supersection
    ns     = descriptor<19>;
    nG     = descriptor<17>;
    s      = descriptor<16>;
    ap     = descriptor<15,11:10>;
    tex    = descriptor<14:12>;
    xn     = descriptor<4>;
    c      = descriptor<3>;
    b      = descriptor<2>;
    pxn    = descriptor<0>;
    domain = '0000';

    walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24));
    walkstate.istable = FALSE;

until walkstate.sdftype != SDFType\_Table;

if afe == '1' && ap<0> == '0' then
    fault.domain      = domain;
    fault.statuscode = Fault\_AccessFlag;

```

```

    return (fault, TTWState UNKNOWN);

// Decode the TEX, C, B and S bits to produce target memory attributes
if tre == '1' then
    walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
elsif RemapRegsHaveResetValues() then
    walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
else
    walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

walkstate.permissions.ap = ap;
walkstate.permissions.xn = xn;
walkstate.permissions.pxn = pxn;
walkstate.domain = domain;
walkstate.nG = nG;

if ss == SS\_Secure && ns == '0' then
    walkstate.baseaddress.paspace = PAS\_Secure;
else
    walkstate.baseaddress.paspace = PAS\_NonSecure;

return (fault, walkstate);

```

### Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```

// AArch32.S2IASize()
// =====
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);

```

### Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```

// AArch32.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);

```



```

// AArch32.S2Walk()
// =====
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams,
                                         AddressDescriptor ipa)
    FaultRecord fault = fault_in;

    if walkparams.sl0 IN {'1x'} || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S2IASize(walkparams.t0sz);
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    if !IsZero(VTTBR<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS_NonSecure;
    baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb));

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    indexmsb = iasize - 1;
    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;
    if HCR2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer indexlsb;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000');

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.tee, walkaddress, fault);

        if fault.statuscode != Fault_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

        case desctype of
            when DescriptorType_Table

```

```

        if !IsZero(descriptor<47:40>) then
            fault.statuscode = Fault_AddressSize;
            return (fault, TTWState UNKNOWN);

        walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12));
        walkstate.level = walkstate.level + 1;
        indexmsb = indexlsb - 1;

    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    when DescriptorType_Page, DescriptorType_Block
        walkstate.istable = FALSE;

until desctype IN {DescriptorType_Page, DescriptorType_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState UNKNOWN);

// Unpack the descriptor into address and upper and lower block attributes
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));

walkstate.permissions.s2ap = descriptor<7:6>;
walkstate.permissions.s2xn = descriptor<54>;
if HaveExtendedExecuteNeverExt() then
    walkstate.permissions.s2xnx = descriptor<53>;
else
    walkstate.permissions.s2xnx = '0';

memattr = descriptor<5:2>;
sh       = descriptor<9:8>;
walkstate.memattrs = S2DecodeMemAttrs(memattr, sh);
walkstate.contiguous = descriptor<52>;

return (fault, walkstate);

```

### Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```

// AArch32.TranslationSizeSD()
// =====
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFTYPE sdftype)
    integer tsize;
    case sdftype of
        when SDFTYPE_SmallPage      tsize = 12;
        when SDFTYPE_LargePage      tsize = 16;
        when SDFTYPE_Section        tsize = 20;
        when SDFTYPE_Supersection   tsize = 24;

    return tsize;

```

### Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```

boolean RemapRegsHaveResetValues();

```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

```
// AArch32.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime_EL2 walkparams = AArch32.S1TTWParamsEL2();
        when Regime_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
        when Regime_EL30 walkparams = AArch32.S1TTWParamsEL30(va);

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx = TGx_4KB;
    walkparams.s = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0 = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGNO;
    walkparams.sh = VTCR.SH0;
    walkparams.ee = HSCTLR.EE;
    walkparams.ptw = HCR.PTW;
    walkparams.vm = HCR.VM OR HCR.DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz<3> then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS);

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

```
// AArch32.GetVARange()
// =====
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    lo_iasize = AArch32.SLIASize(t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.SLIASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange_LOWER else VARange_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange_LOWER;
    elsif IsOnes(va<31:up_iasize>) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange_UNKNOWN;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTL_R_S.C == '1';
        when Regime_EL2  return HSCTLR.C == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTL_R_NS.C else SCTL_R.C) == '1';
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTL_R_S.I == '1';
        when Regime_EL2  return HSCTLR.I == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTL_R_NS.I else SCTL_R.I) == '1';
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL10

```
// AArch32.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLr_Type sctlr;

    if HaveAArch32EL(EL3) then
        ttbcr = TTBCR_NS;
        ttbcr2 = TTBCR2_NS;
        sctlr = SCTLr_NS;
        mair = MAIR1_NS:MAIR0_NS;
        sif = SCR.SIF;
    else
        ttbcr = TTBCR;
        ttbcr2 = TTBCR2;
        sctlr = SCTLr;
        mair = MAIR1:MAIR0;
        sif = SCR_EL3.SIF;

    assert ttbcr.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = ttbcr.T0SZ;
    walkparams.t1sz = ttbcr.T1SZ;
    walkparams.ee = sctlr.EE;
    walkparams.wxn = sctlr.WXN;
    walkparams.uwxn = sctlr.UWXN;
    walkparams.ntlsm = if HaveTrapLoadStoreMultipleDeviceExt() then sctlr.nTlsm else '1';
    walkparams.mair = mair;
    walkparams.sif = sif;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD0 else '0';
    else
        walkparams.sh = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD1 else '0';

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh = HTCR.ORGNO;
    walkparams.hpd = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee = HSCTLR.EE;
    walkparams.wxn = HSCTLR.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = HSCTLR.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// =====
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = TTBCR_S.T0SZ;
    walkparams.t1sz = TTBCR_S.T1SZ;
    walkparams.ee = SCTLR_S.EE;
    walkparams.wxn = SCTLR_S.WXN;
    walkparams.uwxn = SCTLR_S.UWXN;
    walkparams.ntlsmid = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR_S.nTlSMID else '1';
    walkparams.mair = MAIR1_S:MAIR0_S;
    walkparams.sif = SCR.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD0 else '0';
    else
        walkparams.sh = TTBCR_S.SH1;
        walkparams.irgn = TTBCR_S.IRGN1;
        walkparams.orgn = TTBCR_S.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';

    return walkparams;
```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress,
                                AccType acctype, integer size)
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumBreakpointsImplemented\(\);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT IN {'0x01'};
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                    linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAArch32\(\) && size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool\(Unpredictable\_BPMATCHHALF\);
    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool\(Unpredictable\_BPMATCHHALF\);

    match = value_match && state_match && enabled;

    return match;
```



```

// AArch64.BreakpointValueMatch()
// =====

boolean AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        Constraint c;
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMP);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR_EL1[n].BT;
    Constraint c;

    if ((dbgtype IN {'011x', '11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
        dbgtype IN {'010x'} || // Reserved
        (! (dbgtype IN {'0x0x'}) && !context_aware) || // Context match
        (dbgtype IN {'1xxx'} && !HaveEL(EL2))) then // EL2 extension
        (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (dbgtype IN {'0x0x'});
    match_vmid = (dbgtype IN {'10xx'});
    match_cid = (dbgtype IN {'001x'});
    match_cid1 = (dbgtype IN {'101x', 'x11x'});
    match_cid2 = (dbgtype IN {'11xx'});
    linked = (dbgtype IN {'xxx1'});

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, or if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    boolean BVR_match;
    if match_addr then
        boolean byte_select_match;
        byte = UInt(vaddress<1:0>);
        if HaveAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2}; // "vaddress" is halfword aligned
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0; // "vaddress" is word aligned
            byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
        // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
        // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access

```

```

// generates a Translation fault.
integer top = AArch64.VAMax();
if !IsOnes(DBGGBVR_EL1[n]<63:top>) && !IsZero(DBGGBVR_EL1[n]<63:top>) then
    if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
        top = 63;
BVR_match = (vaddress<top:2> == DBGGBVR_EL1[n]<top:2>) && byte_select_match;

elseif match_cid then
    if IsInHost() then
        BVR_match = (CONTEXTIDR_EL2<31:0> == DBGGBVR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGGBVR_EL1[n]<31:0>);
elseif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGGBVR_EL1[n]<31:0>);
boolean BXVR_match;
if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;
    if !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGGBVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGGBVR_EL1[n]<47:32>;
    BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        !IsInHost() &&
        vmid == bvr_vmid);
elseif match_cid2 then
    BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug()) &&
        EL2Enabled() &&
        DBGGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

return match;

```



```

// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC_in, bit HMC_in,
                           bits(2) PxC_in, boolean linked_in, bits(4) LBN,
                           boolean isbreakpt, AccType acctype, boolean ispriv)

// "SSC_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
bits(2) SSC = SSC_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpt);
if c == Constraint\_DISABLED then return FALSE;
// Otherwise the HMC,SSC,PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.

EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

boolean priv_match;
if HaveNV2Ext() && acctype == AccType\_NV2REGISTER && !isbreakpt then
    priv_match = EL2_match;
elsif !ispriv && !isbreakpt then
    priv_match = EL0_match;
else
    case PSTATE.EL of
        when EL3 priv_match = EL3_match;
        when EL2 priv_match = EL2_match;
        when EL1 priv_match = EL1_match;
        when EL0 priv_match = EL0_match;

boolean security_state_match;
ss = CurrentSecurityState();
case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = ss == SS\_NonSecure; // Non-secure only
    when '10' security_state_match = ss == SS\_Secure; // Secure only
    when '11' security_state_match = (HMC == '1' || ss == SS\_Secure); // HMC=1 -> Both, 0 -> Secure
integer lbn;
if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
    last_ctx_cmp = NumBreakpointsImplemented() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable\_BPNOTCTX);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
        case c of
            when Constraint\_DISABLED return FALSE; // Disabled
            when Constraint\_NONE linked = FALSE; // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

boolean linked_match;
if linked then
    vaddress = bits(64) UNKNOWN;
    linked_to = TRUE;
    linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

```

```
return priv_match && security_state_match && (!linked || linked_match);
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    ss = return CurrentSecurityStateAArch64.GenerateDebugExceptionsFrom();
    return(PSTATE.EL, AArch64.GenerateDebugExceptionsFromIsSecure(PSTATE.EL, ss, PSTATE.D);(), PSTATE.D);
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, AArch64.GenerateDebugExceptionsFrom(bits(2) from_el,
    if OSLSR_EL1.OSLK == '1' || SecurityState from_state, bit mask)
    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = ( route_to_el2 = HaveEL(EL2) && (from_state !=) && (!secure || SS_Secure || IsSecure
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    () && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'););
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
) && secure then
        enabled = MDCR_EL3.SDD == '0';
    if from_el == if from == EL0 && ELUsingAArch32(EL1) then
        enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);
(from);

    return enabled;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch64.CheckForPMUOverflow()
    boolean pmuirq;
    bit E;
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1.C == '1' && PMOVSSET_EL0.C == '1';
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HPME else PMCR_EL0.E;
            if E == '1' && PMINTENSET_EL1<idx> == '1' && PMOVSSET_EL0<idx> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMIUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSLR_EL0.)
```

## Library pseudocode for aarch64/debug/pmu/AArch64.ClearEventCounters

```
// AArch64.ClearEventCounters()
// =====
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_EL0[idx] = Zeros\(\);
```



```

// AArch64.CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.
// Return TRUE if counter "idx" should count its event. For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch64.CountPMUEvents(integer idx)
    assert idx == CYCLE_COUNTER_ID || idx < GetNumEventCounters();
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch64.PMUCounterIsHyp(idx);
    ss =
    // Main enable controls
    if idx == CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
        enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
    else
        E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
        enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * MDCR_EL3.SPME == 0, and either:
    //   - FEAT_PMUv3p7 is not implemented
    //   - MDCR_EL3.MPMX == 0
    if HaveEL(EL3) && ss == SS_SecureIsSecure then
    () then
        if HavePMUv3p7() then
            prohibited = MDCR_EL3.<SPME,MPMX> == '00';
        else
            prohibited = MDCR_EL3.SPME == '0';

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMUv3p7 is implemented
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0
    //   - PMNx is not reserved for EL2
    // * MDCR_EL3.MPMX == 1
    if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
        prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = MDCR_EL2.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen
    frozen = FALSE;

```

```

// If FEAT_PMuV3p7 is implemented, event counting can be frozen
if HavePMuV3p7() then
    ovflws = ZeroExtend(PMOVSSET_EL0<GetNumEventCounters()-1:0>);
    bit FZ;
    if resvd_for_el2 then
        FZ = MDCR_EL2.HPMFZ0;
        ovflws<UInt(MDCR_EL2.HPMN)-1:0> = Zeros();
    else
        FZ = PMCR_EL0.FZ0;
        if HaveEL(EL2) && UInt(MDCR_EL2.HPMN) < GetNumEventCounters() then
            ovflws<GetNumEventCounters()-1:UInt(MDCR_EL2.HPMN)> = Zeros();
        frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR_EL0.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    enabled = enabled && (PMCR_EL0.DP == '0');
    // Otherwise whether event counting is prohibited does not affect the cycle counter
    prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMuV3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if HavePMuV3p5() && idx == CYCLE_COUNTER_ID then
    if (ifHaveEL(EL3) && ss == SS_Secure && MDCR_EL3.SCCD == '1') then
        prohibited = TRUE;
    if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
        prohibited = TRUE;

// If FEAT_PMuV3p7 is implemented, cycle counting can be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if HavePMuV3p7() && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR_EL0<31:0> else PMEVTYPER_EL0[idx]<31:0>;

P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
M = if HaveEL(EL3) then filter<26> else '0';
SH = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
    when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
    when EL2 filtered = if ss == SS_Secure then NSH == SH else NSH == '0';
    when EL3 filtered = M != P;

return !debug && enabled && !prohibited && !filtered && !frozen;

```

## Library pseudocode for aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

```
// AArch64.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters\(\);
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled\(\) then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!HaveFeatHPMN0\(\) && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable\_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    if HavePMUv3p5\(\) then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        lp = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
        ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>);
        ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
        PMOVSLR_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters\(\) && (!HavePMUv3p5\(\) || lp == '0') then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

## Library pseudocode for aarch64/debug/pmu/AArch64.PMUCounterIsHyp

```
// AArch64.PMUCounterIsHyp
// =====
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch64.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE\_COUNTER\_ID && HaveEL(EL2) then
        resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN);
        if UInt(MDCR_EL2.HPMN) > GetNumEventCounters\(\) || (!HaveFeatHPMN0\(\) && IsZero(MDCR_EL2.HPMN)) then
            resvd_for_el2 = boolean UNKNOWN;
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.PMUCycle

```
// AArch64.PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if AArch64.CountPMUEvents(idx) then
                accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter(idx, accumulated);
                PMUEventAccumulator[idx] = 0;

    integer old_value;
    integer new_value;
    integer ovflw;
    if (AArch64.CountPMUEvents(CYCLE_COUNTER_ID) &&
        (!HaveAArch32() || PMCR_EL0.LC == '1' || PMCR_EL0.D == '0' || HasElapsed64Cycles())) then
        old_value = UInt(PMCCNTR_EL0);
        new_value = old_value + 1;
        PMCCNTR_EL0 = new_value<63:0>;

        if HaveAArch32() then
            ovflw = if PMCR_EL0.LC == '1' then 64 else 32;
        else
            ovflw = 64;

        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET_EL0.C = '1';
            PMOVSCLR_EL0.C = '1';

    AArch64.CheckForPMUOverflow();
```

## Library pseudocode for aarch64/debug/pmu/AArch64.PMUSwIncrement

```
// AArch64.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC_EL0.

AArch64.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// =====

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// =====

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS\_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;

    bits(2) PCT_el1;
    if !HaveECVExt() then
        PCT_el1 = '0':PMSCR_EL1.PCT<0>; // PCT<1> is RES0
    else
        PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
    if EL2Enabled() then
        bits(2) PCT_el2;
        if !HaveECVExt() then
            PCT_el2 = '0':PMSCR_EL2.PCT<0>; // PCT<1> is RES0
        else
            PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
    case PCT_el2 of
        when '00'
            return if return IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
        when '01'
            if owning_el == EL2 then return TimeStamp_Physical;
        when '11'
            assert HaveECVExt(); // FEAT_ECV must be implemented
            if owning_el == EL1 && PCT_el1 == '00' then
                return if return IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
            else
                return TimeStamp_OffsetPhysical;
        otherwise
            Unreachable();

    case PCT_el1 of
        when '00' return if IsInHost() then TimeStamp_Physical else();

case PCT_el1 of
    when '00' return TimeStamp_Virtual;
    when '01' return TimeStamp_Physical;
    when '11'
        assert HaveECVExt(); // FEAT_ECV must be implemented
        return TimeStamp_OffsetPhysical;
    otherwise Unreachable();
```

## Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
    OpType_Load, // Any memory-read operation other than atomics, compare-and-swap, and swap
    OpType_Store, // Any memory-write operation, including atomics without return
    OpType_LoadAtomic, // Atomics with return, compare-and-swap and swap
    OpType_Branch, // Software write to the PC
    OpType_Other // Any other class of operation
};
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    state_match = ((owning_ss == SS_Secure    && SCR_EL3.NS == '0') ||
                  (owning_ss == SS_NonSecure && SCR_EL3.NS == '1'));
    return (!ELUsingAArch32(owning_el) && state_match &&
           PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

    if HaveEL(EL3) then
        owning_ss = if MDCR_EL3.NSPB<1> == '0' then SS_Secure else SS_NonSecure;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;

    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
    else
        owning_el = EL1;

    return (owning_ss, owning_el);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

Library pseudocode for aarch64/debug/  
statisticalprofiling/**StatisticalProfilingEnabled****SPECollectRecord**

```

// StatisticalProfilingEnabled()
// =====
// SPECollectRecord()
// =====
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean StatisticalProfilingEnabled()
    if !SPECollectRecord(bits(64) events, integer total_latency, HaveStatisticalProfilingOpType() || optype
    assert UsingAArch32StatisticalProfilingEnabled() || !();

    bits(64) mask = 0xAA<63:0>; // Bits [7,5,3,1]
    bits(64) e;
    bits(64) m;
    if ProfilingBufferEnabledHaveStatisticalProfilingv1p1() then
        return FALSE;

    tge_set =() then mask<11> = '1'; // Alignment Flag
    if EL2EnabledHaveStatisticalProfilingv1p2() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) =() then mask<6> = '1'; // Not taken flag

    mask<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE mask 63:48";
    mask<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE mask 31:24";
    mask<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE mask 15:12";

    // Check for UNPREDICTABLE case
    if ( ProfilingBufferOwnerHaveStatisticalProfilingv1p2();
    if ( && PMSFCR_EL1.<FnE,FE> == '11' &&
        !UIntIsZero(owning_el) <(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
        if ConstrainUnpredictableBool(Unpredictable_BADPMSFCR) then
            return FALSE;
    else
        // Filtering by event
        if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
            e = events AND mask;
            m = PMSEVFR_EL1 AND mask;
            if !IsZero(NOT(e) AND m) then return FALSE;

        // Filtering by inverse event
        if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.FnE == '1' &&
            !IsZero(PMSNEVFR_EL1)) then
            e = events AND mask;
            m = PMSNEVFR_EL1 AND mask;
            if !IsZero(e AND m) then return FALSE;

    // Filtering by type
    if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
        case optype of
            when OpType_Branch
                if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType_Load
                if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType_Store
                if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType_LoadAtomic
                if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise
                return FALSE;

    // Filtering by latency
    if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
        if total_latency < UInt(PSTATE.EL) || (tge_set && owning_el == (PMSLATFR_EL1.MINLAT) then
            return FALSE;

    // Check for UNPREDICTABLE cases
    if ((PMSFCR_EL1.FE == '1' && EL1IsZero) ||
        owning_ss != (PMSEVFR_EL1 AND mask)) ||
        (PMSFCR_EL1.FT == '1' && CurrentSecurityStateIsZero()) then
        return FALSE;
    bit spe_bit;

```

```

case PSTATE.EL of
  when (PMSFCR_EL1.<B,LD,ST>)) ||
    (PMSFCR_EL1.FL == '1' && EL3IsZero (PMSLATFR_EL1.MINLAT))) then
    return UnreachableConstrainUnpredictableBool();
  when (EL2Unpredictable_BADPMSFCR spe_bit = PMSCR_EL2.E2SPE;
  when);

  if ( EL1HaveStatisticalProfilingv1p2 spe_bit = PMSCR_EL1.E1SPE;
  when() &&
    ((PMSFCR_EL1.FnE == '1' && (PMSNEVFR_EL1 AND mask)) ||
    (PMSFCR_EL1.<FnE,FE> == '11' &&
    !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)))) then
    return ConstrainUnpredictableBool(Unpredictable_BADPMSFCREL0IsZero spe_bit = (if tge_set then PM
);

return spe_bit == '1'; return TRUE;

```

## Library pseudocode for aarch64/debug/statisticalprofiling/TimeStampStatisticalProfilingEnabled

```

enumeration// StatisticalProfilingEnabled()
// =====

boolean TimeStamp {StatisticalProfilingEnabled()
  if !
    TimeStamp_None, // No timestamp() ||
    TimeStamp_CoreSight, // CoreSight time (IMPLEMENTATION DEFINED)() || !
    TimeStamp_Physical, // Physical counter value with no offset() then
    return FALSE;

  tge_set =
    TimeStamp_OffsetPhysical, // Physical counter value minus CNTPOFF_EL2() && HCR_EL2.TGE == '1';
  (owning_ss, owning_el) =
  ();
  if (UInt(owning_el) < UInt(PSTATE.EL) || (tge_set && owning_el == EL1) ||
    owning_ss != CurrentSecurityState()) then
    return FALSE;
  bit spe_bit;
  case PSTATE.EL of
    when EL3 Unreachable();
    when EL2 spe_bit = PMSCR_EL2.E2SPE;
    when EL1 spe_bit = PMSCR_EL1.E1SPE;
    when EL0TimeStamp_Virtual }; // Physical counter value minus CNTVOFF_EL2spe_bit = (if tge

return spe_bit == '1';

```

## Library pseudocode for aarch64/

### debug/takeexceptiondbgstatisticalprofiling/AArch64.TakeExceptionInDebugStateTimeStamp

```
// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64.enumeration

AArch64.TakeExceptionInDebugState(bits(2) target_el, TimeStamp { ExceptionRecord exception_in }
    assertTimeStamp_None, // No timestamp HaveEL(target_el) && !TimeStamp_CoreSight,
    ExceptionRecord exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if HaveIESB() then
        sync_errors = SCTLRL[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // SCTLRL[].IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(64) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLRL[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUA0Ext() then PSTATE.UA0 = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEEExt() then PSTATE.TC0 = '1';

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction(); TimeStamp_Virtual }; // Physical counter value minus CNTVOFF_EL2
```

Library pseudocode for aarch64/

debug/watchpointtakeexceptiondbg/AArch64.WatchpointByteMatchAArch64.TakeExceptionInDebugS

```

// AArch64.WatchpointByteMatch()
// =====

boolean// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64. AArch64.WatchpointByteMatch(inte

integer top =exception_in)
assert AArch64.VAMaxHaveEL();
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
byte_select_match = (DBGWCR_EL1[n].BAS<(target_el) && !ELUsingAArch32(target_el) && UInt(vaddress<bot
mask =(target_el) >= UInt(DBGWCR_EL1[n].MASK);

// If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
// DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !(PSTATE.EL);IsOnesExceptionRecord(DBGWCR_EL1[n].BAS) then
    byte_select_match =exception = exception_in;
boolean sync_errors;
boolean iesb_req;
if ConstrainUnpredictableBoolHaveIESB(()) then
    sync_errors =Unpredictable_WPMASKANDBASSCTLR);
else
    LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB = (DBGWCR_EL1[n].BAS + LSB);
    if ![target_el].IESB == '1';
    ifIsZeroHaveDoubleFaultExt(MSB AND (MSB - 1)) then // Not contiguous
        byte_select_match =() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // SCTLRL[.IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUSUnpredictable_IESBinDebug);
        bottom = 3; // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then) then
    sync_errors = FALSE;
else
    sync_errors = FALSE;
    ConstrainSynchronizeContext c;
    (c, mask) =();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = ConstrainUnpredictableIntegerUsingAArch32(3, 31,());
if from_32 then Unpredictable_RESWPMASKAArch64.MaybeZeroRegisterUppers);
assert c IN {()};Constraint_DISABLEDAArch64.ReportException,(exception, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1'; Constraint_NONESPSR,[] = bits(64) UNKNOWN; Constraint_UNKNOWNELR};
case c of
    when[] = bits(64) UNKNOWN;

// PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0'; // PSTATE.J is RES0
if ( Constraint_DISABLEDHavePANExt return FALSE; // Disabled
    when() && (PSTATE.EL == Constraint_NONEEL1 mask = 0; // No masking
    // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

boolean WVR_match;
if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    if !((PSTATE.EL == IsOnesEL2(DBGBVR_EL1[n]<63:top>) && !&&IsZeroELIsInHost(DBGBVR_EL1[n]<63:top>
        if( ConstrainUnpredictableBoolEL0(()) && Unpredictable_DBGxVR_RESSSCTLR) then
            top = 63;
    WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);

```

```

// If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
if WVR_match && ![] .SPAN == '0') then
    PSTATE.PAN = '1';
if IsZeroHaveUA0Ext(DBGWVR_EL1[n]<mask-1:bottom>) then
    WVR_match =() then PSTATE.UAO = '0';
if ConstrainUnpredictableBoolHaveBTIExt() then PSTATE.BTYPE = '00';
if() then PSTATE.SSBS = bit UNKNOWN;
if HaveMTEExt() then PSTATE.TCO = '1';

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;

EDSCR.ERR = '1';
UpdateEDSCRFields(); // Update EDSCR processor state flags.

if sync_errors then
    SynchronizeErrors();

EndOfInstructionUnpredictable_WPMASKEDBITSHaveSSBSExt);
else
    WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

return WVR_match && byte_select_match;();

```

Library pseudocode for aarch64/debug/  
watchpoint/**AArch64.WatchpointMatch****AArch64.WatchpointByteMatch**

```

// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.
// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv, AArch64.WatchpointMatch
                                AccType acctype, boolean iswrite)
    assert !acctype, bits(64) vaddress)

    integer top = ELUsingAArch32AArch64.VAMax();
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<S1TranslationRegimeUInt());
    assert n <(vaddress<bottom-1:0>)> != '0');
    mask = NumWatchpointsImplementedUInt();
    (DBGWCR_EL1[n].MASK);

    // "ispriv" is:
    // * FALSE for all loads, stores, and atomic operations executed at EL0.
    // * FALSE if the access is unprivileged.
    // * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '111'
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && ! AArch64.StateMatchIsOnes(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                                linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);

    ls_match = FALSE;
    if(DBGWCR_EL1[n].BAS) then
        byte_select_match = IsAtomicRWConstrainUnpredictableBool(acctype) then
            ls_match = (DBGWCR_EL1[n].LSC != '00');
    elseif iswrite then
        ls_match = (DBGWCR_EL1[n].LSC<1> != '0');
    else
        ls_match = (DBGWCR_EL1[n].LSC<0> != '0');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || ( );
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3; // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE; // Disabled
            when Constraint_NONE mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR_match;
    if mask > bottom then
        // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
        // of DBGxVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        if !IsOnes(DBGxVR_EL1[n]<63:top>) && !IsZero(DBGxVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then

```

```

WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITSAArch64.WatchpointByteMatch
);
else
WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

return value_match && state_match && ls_match && enabled; return WVR_match && byte_select_match;

```

## Library pseudocode for

### aarch64/exceptionsdebug/abortswatchpoint/AArch64.AbortAArch64.WatchpointMatch

```

// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime. // AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean

AArch64.Abort(bits(64) vaddress, AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean
    ifacctype, boolean iswrite)
assert ! IsDebugExceptionELUsingAArch32(fault) then
    if fault.acctype == ( AccType_IFETCHS1TranslationRegime then
        if();
assert n < UsingAArch32NumWatchpointsImplemented() && fault.debugmoe ==();

// "ispriv" is:
// * FALSE for all loads, stores, and atomic operations executed at EL0.
// * FALSE if the access is unprivileged.
// * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR_EL1[n].E == '1';
linked = DBGWCR_EL1[n].WT == '1';
isbreakpnt = FALSE;

state_match = DebugException_VectorCatchAArch64.StateMatch then(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC,
    linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
ls_match = FALSE;
if acctype ==
    AArch64.VectorCatchExceptionAccType_ATOMICRW(fault);
    elsethen
        ls_match = (DBGWCR_EL1[n].LSC != '00');
    else
        ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

value_match = FALSE;
for byte = 0 to size - 1
    value_match = value_match ||
        AArch64.BreakpointExceptionAArch64.WatchpointByteMatch(fault);
    else
        AArch64.WatchpointException(vaddress, fault);
elseif fault.acctype == AccType_IFETCH then
    AArch64.InstructionAbort(vaddress, fault);
else
    AArch64.DataAbort(vaddress, fault);(n, acctype, vaddress + byte);

return value_match && state_match && ls_match && enabled;

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndromeAArch64.Abort

```
// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime. AArch64.AbortSyndrome(AArch64.Abort
exception = fault)

if ExceptionSyndromeIsDebugException(exceptype);

d_side = exceptype IN {(fault) then
    if fault.acctype == Exception_DataAbortAccType_IFETCH, then
        if Exception_NV2DataAbortUsingAArch32,() && fault.debugmoe == Exception_WatchpointDebugExcept

(exception.syndrome, exception.syndrome2) =(fault);
    else AArch64.FaultSyndromeAArch64.BreakpointException(d_side, fault);
exception.vaddress =(fault);
    else ZeroExtendAArch64.WatchpointException(vaddress);
if(vaddress, fault);
elseif fault.acctype == IPAValidAccType_IFETCH(fault) then
    exception.ipavalid = TRUE;
    exception.NS = if fault.ipaddress.paspace == then (vaddress, fault);
else
    AArch64.DataAbortPAS_NonSecureAArch64.InstructionAbort then '1' else '0';
    exception.ipaddress = fault.ipaddress.address;
else
    exception.ipavalid = FALSE;

return exception;(vaddress, fault);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignmentAArch64.AbortSyndrome

```
// AArch64.CheckPCAlignment()
// =====// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord

AArch64.CheckPCAlignment()

bits(64) pc =AArch64.AbortSyndrome( ThisInstrAddrException());
if pc<1:0> != '00' thenexceptype,
    fault, bits(64) vaddress)
exception = ExceptionSyndrome(exceptype);

d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_

(exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
exception.vaddress = ZeroExtend(vaddress);
if IPAValid(fault) then
    exception.ipavalid = TRUE;
    exception.NS = if fault.ipaddress.paspace == PAS_NonSecureAArch64.PCAlignmentFaultFaultRecord();
    exception.ipaddress = fault.ipaddress.address;
else
    exception.ipavalid = FALSE;

return exception;
```

## Library pseudocode for aarch64/exceptions/ aborts/AArch64.DataAbortAArch64.CheckPCAlignment

```
// AArch64.DataAbort()
// =====// AArch64.CheckPCAlignment()
// =====

AArch64.DataAbort(bits(64) vaddress, AArch64.CheckPCAlignment())

bits(64) pc = FaultRecord fault)
route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
    (HCR_EL2.TGE == '1' ||
    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
    (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) ||
    IsSecondStage(fault)));

bits(64) preferred_exception_return = ThisInstrAddr();
integer vect_offset;
if ( (if pc<1:0> != '00' then HaveDoubleFaultExtAArch64.PCAlignmentFault() && (PSTATE.EL == EL3 ||
    IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
    vect_offset = 0x180;
else
    vect_offset = 0x0;
ExceptionRecord exception;
if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
    exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
else
    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
bits(2) target_el = EL1;
if PSTATE.EL == EL3 || route_to_el3 then
    target_el = EL3;
elsif PSTATE.EL == EL2 || route_to_el2 then
    target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);()
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCFAArch64.DataAbort

```
// AArch64.EffectiveTCF()
// =====
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) // AArch64.DataAbort()
// ===== AArch64.EffectiveTCF(AArch64.DataAbort(bits(64) vaddress, AccType FaultRecord acctype)
    bits(2) tcf, el;
    el = fault;
    route_to_el3 = S1TranslationRegimeHaveEL();

    if el == EL3 then
        tcf = SCTL_EL3.TCF;
    elsif el == EL2 && SCR_EL3.EA == '1' && EL2IsExternalAbort then
        if(fault);
        route_to_el2 = ( AArch64.AccessUsesEL2Enabled(acctype) == () && PSTATE.EL IN { EL0 } then
            tcf = SCTL_EL2.TCF0;
        else
            tcf = SCTL_EL2.TCF;
    elsif el == EL1 then
        if} &&
        (HCR_EL2.TGE == '1' ||
        ( AArch64.AccessUsesELHaveRASExt(acctype) == () && HCR_EL2.TEA == '1' && EL0IsExternalAbort
            tcf = SCTL_EL1.TCF0;
        else
            tcf = SCTL_EL1.TCF;

    if tcf == '11' then //reserved value
        if !(fault)) ||
        (HaveMTE3ExtHaveNV2Ext() then
            (-, tcf) = () && fault.acctype == ConstrainUnpredictableBitsAccType_NV2REGISTER() || (fault));

    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    ExceptionRecord exception;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    bits(2) target_el = EL1;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    AArch64.TakeExceptionUnpredictable_RESTCFIsSecondStage);

    return tcf; (target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ aborts/AArch64.InstructionAbortAArch64.EffectiveTCF

```
// AArch64.InstructionAbort()
// =====// AArch64.EffectiveTCF()
// =====
// Returns the TCF field applied to tag check faults in the given Exception level.
bits(2)

AArch64.InstructionAbort(bits(64) vaddress, AArch64.EffectiveTCF( FaultRecordAccType fault)
// External aborts on instruction fetch must be taken synchronously
if(acctype)
bits(2) tcf, el;
el = HaveDoubleFaultExtSITranslationRegime() then assert fault.statuscode !=();

if el == Fault_AsyncExternal;
route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && then
tcf = SCTLR_EL3.TCF;
elseif el == IsExternalAbortEL2(fault);
route_to_el2 = (then
ifEL2EnabledAArch64.AccessUsesEL() && PSTATE.EL IN {(acctype) == EL0, then
tcf = SCTLR_EL2.TCF0;
else
tcf = SCTLR_EL2.TCF;
elseif el == EL1} &&
(HCR_EL2.TGE == '1' ||
(then
ifHaveRASExtAArch64.AccessUsesEL() && HCR_EL2.TEA == '1' &&(acctype) == IsExternalAbortEL0(fault);
tcf = SCTLR_EL1.TCF0;
else
tcf = SCTLR_EL1.TCF;

if tcf == '11' then //reserved value
if !
IsSecondStageHaveMTE3Ext(fault));() then
(-, tcf) =

ExceptionRecordConstrainUnpredictableBits exception;
bits(64) preferred_exception_return = ( ThisInstrAddrUnpredictable_RESTCF());
integer vect_offset;
if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
vect_offset = 0x180;
else
vect_offset = 0x0;

exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

bits(2) target_el = EL1;
if PSTATE.EL == EL3 || route_to_el3 then
target_el = EL3;
elseif PSTATE.EL == EL2 || route_to_el2 then
target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset););

return tcf;
```

## Library pseudocode for aarch64/exceptions/ aborts/AArch64.PCAlignmentFaultAArch64.InstructionAbort

```
// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.// AArch64.InstructionAbort()
// =====

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = AArch64.InstructionAbort(bits(64) vaddress, ThisInstrAddrFaultRe
    vect_offset = 0x0;

    exception = fault;
    // External aborts on instruction fetch must be taken synchronously
    if ExceptionSyndromeHaveDoubleFaultExt() then assert fault.statuscode != Exception_PCAlignmentFault_
    exception.vaddress = ;
    route_to_el3 = ThisInstrAddrHaveEL();

    bits(2) target_el = ( EL3 ) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = ( EL2Enabled() && PSTATE.EL IN {EL0, EL1;
    if} &&
        (HCR_EL2.TGE == '1' ||
        ( UIntHaveRASExt(PSTATE.EL) >() && HCR_EL2.TEA == '1' && UIntIsExternalAbort((fault)

    ExceptionRecord exception;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    bits(2) target_el = EL1; then
        target_el = PSTATE.EL;
    elsif;
    if PSTATE.EL == || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2EL2EnabledEL3() && HCR_EL2.TGE == '1' then
    || route_to_el2 then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ aborts/~~AArch64.RaiseTagCheckFault~~~~AArch64.PCAlignmentFault~~

```
// AArch64.RaiseTagCheckFault()  
// =====  
// Raise a tag check fault exception.// AArch64.PCAlignmentFault()  
// =====  
// Called on unaligned program counter in AArch64 state.  
  
AArch64.RaiseTagCheckFault(bits(64) va, boolean write)  
AArch64.PCAlignmentFault()  
  
    bits(64) preferred_exception_return = ThisInstrAddr();  
    integer vect_offset = 0x0;  
vect_offset = 0x0;  
  
    exception = ExceptionSyndrome(Exception_DataAbortException_PCAlignment);  
    exception.syndrome<5:0> = '010001';  
    if write then  
        exception.syndrome<6> = '1';  
    exception.vaddress = bits(4) UNKNOWN : va<59:0>;  
  
bits(2) target_el = exception.vaddress = ThisInstrAddr();  
  
bits(2) target_el = EL1;  
    if UInt(PSTATE.EL) > UInt(EL1) then  
        target_el = PSTATE.EL;  
    elsif PSTATE.EL == EL0 && then  
target_el = PSTATE.EL;  
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then  
        target_el = EL2;  
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ aborts/AArch64.ReportTagCheckFaultAArch64.RaiseTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =====
// Records a tag check fault exception into the appropriate TCFR_ELx. // AArch64.RaiseTagCheckFault()
// =====
// Raise a tag check fault exception.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == AArch64.RaiseTagCheckFault(bits(64) va, boolean write)
        bits(64) preferred_exception_return = EL3ThisInstrAddr then
            assert ttbr == '0';
            TFSR_EL3.TF0 = '1';
        elsif el == ();
            integer vect_offset = 0x0;

            exception = EL2ExceptionSyndrome then
                if ttbr == '0' then
                    TFSR_EL2.TF0 = '1';
                else
                    TFSR_EL2.TF1 = '1';
            elsif el == ( Exception_DataAbort );
                exception.syndrome<5:0> = '010001';
            if write then
                exception.syndrome<6> = '1';
            exception.vaddress = bits(4) UNKNOWN : va<59:0>;

            bits(2) target_el = EL1 then
                if ttbr == '0' then
                    TFSR_EL1.TF0 = '1';
                else
                    TFSR_EL1.TF1 = '1';
            elsif el == ;
                if UInt(PSTATE.EL) > UInt(EL1) then
                    target_el = PSTATE.EL;
                elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
                    target_el = EL2;
                AArch64.TakeException then
                    if ttbr == '0' then
                        TFSRE0_EL1.TF0 = '1';
                    else
                        TFSRE0_EL1.TF1 = '1'; (target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ aborts/AArch64.SPAlignmentFaultAArch64.ReportTagCheckFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.// AArch64.ReportTagCheckFault()
// =====
// Records a tag check fault exception into the appropriate TCFR_ELx.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == ThisInstrAddrEL3();
    vect_offset = 0x0;

    exception = then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elsif el == ExceptionSyndromeEL2(then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elsif el == Exception_SPAlignment);

    bits(2) target_el = EL1;
    if then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elsif el == UIntEL0(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset); then
        if ttbr == '0' then
            TFSRE0_EL1.TF0 = '1';
        else
            TFSRE0_EL1.TF1 = '1';
```

## Library pseudocode for aarch64/exceptions/ aborts/AArch64.TagCheckFaultAArch64.SPAlignmentFault

```
// AArch64.TagCheckFault()
// =====
// Handle a tag check fault condition.// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.TagCheckFault(bits(64) vaddress, AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = AccTypeThisInstrAddr acctype, boolean iswrite)
    bits(2) tcf, el;
    el = ();
    vect_offset = 0x0;

    exception = AArch64.AccessUsesELExceptionSyndrome(acctype);
    tcf = ( AArch64.EffectiveTCFException_SPAlignment(acctype);
    case tcf of
        when '00' // Tag Check Faults have no effect on the PE
            return;
        when '01' // Tag Check Faults cause a synchronous exception);

    bits(2) target_el =
        AArch64.RaiseTagCheckFaultEL1(vaddress, iswrite);
    when '10' // Tag Check Faults are asynchronously accumulated;
    if
        AArch64.ReportTagCheckFaultUInt(el, vaddress<55>);
    when '11' // Tag Check Faults cause a synchronous exception on reads or on
        // a read/write access, and are asynchronously accumulated on writes
        // Check for access performing both a read and a write.
        if !iswrite || (PSTATE.EL) -> IsAtomicRWUInt(acctype) then(
            AArch64.RaiseTagCheckFaultEL1(vaddress, iswrite);
        else) then
            target_el = PSTATE.EL;
    elsif
        () && HCR_EL2.TGE == '1' then
            target_el = EL2;
    AArch64.TakeExceptionAArch64.ReportTagCheckFaultEL2Enabled(PSTATE.EL, vaddress<55>);(target_el, excep
```

## Library pseudocode for aarch64/exceptions/ aborts/**BranchTargetException****AArch64.TagCheckFault**

```
// BranchTargetException()
// AArch64.TagCheckFault()
// =====
// Raise branch target exception.// Handle a tag check fault condition.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return =AArch64.TagCheckFault(bits(64) vaddress, ThisInstrAddrAccType(),
    vect_offset = 0x0;

    exception =acctype, boolean iswrite)
    bits(2) tcf, el;
    el = ExceptionSyndromeAArch64.AccessUsesEL((acctype);
    tcf =Exception_BranchTargetAArch64.EffectiveTCF);
    exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> =(acctype);
    case tcf of
        when '00' // Tag Check Faults have no effect on the PE
            return;
        when '01' // Tag Check Faults cause a synchronous exception ZerosAArch64.RaiseTagCheckFault

    bits(2) target_el =(vaddress, iswrite);
    when '10' // Tag Check Faults are asynchronously accumulated EL1AArch64.ReportTagCheckFault
    if(el, vaddress<55>);
        when '11' // Tag Check Faults cause a synchronous exception on reads or on
            // a read-write access, and are asynchronously accumulated on writes
            // Check for access performing both a read and a write.
            readwrite = acctype IN { UIntAccType_ATOMICRW(PSTATE.EL) >, UIntAccType_ORDEREDATOMICRW(,EL1
            target_el = PSTATE.EL;
        elsif PSTATE.EL ==};

        if !iswrite || readwrite then EL0AArch64.RaiseTagCheckFault &&(vaddress, iswrite);
        else EL2EnabledAArch64.ReportTagCheckFault() && HCR_EL2.TGE == '1' then
            target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);(PSTATE.EL, vad
```

## Library pseudocode for aarch64/

### exceptions/asyncaorts/AArch64.TakePhysicalFIQExceptionBranchTargetException

```
// AArch64.TakePhysicalFIQException()
// =====// BranchTargetException()
// =====
// Raise branch target exception.

AArch64.TakePhysicalFIQException()

    route_to_el3 = AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_FIQException_BranchTarget);

    if route_to_el3 then exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> =
        AArch64.TakeExceptionZeros(); // RES0

    bits(2) target_el = EL3EL1, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL ==;
    if EL2UInt || route_to_el2 then
        assert PSTATE.EL != (PSTATE.EL) -> EL3UInt;
        AArch64.TakeExceptionEL1() then
            target_el = PSTATE.EL;
    elsif PSTATE.EL == EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, && EL1EL2Enabled}; () && HCR_EL2.TGE == '1' then
            target_el =
                EL2;
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset); (target_el, exception,
```

## Library pseudocode for aarch64/exceptions/

### asyncaorts/AArch64.TakePhysicalIRQExceptionAArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalIRQException()
AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    ) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
        (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_IRQException_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/

### async/AArch64.TakePhysicalSErrorExceptionAArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalSErrorException()
// =====// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalSErrorException(bits(25) syndrome)
AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || (!
            (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1')
        bits(64) preferred_exception_return = IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;
    vect_offset = 0x80;

    bits(2) target_el;
    if PSTATE.EL == exception = EL3ExceptionSyndrome || route_to_el3 then
        target_el = (Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3;
, exception, preferred_exception_return, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_el2 then
        target_el = assert PSTATE.EL != EL2EL3;
    else
        target_el =; EL1AArch64.TakeException;

    if( IsSErrorEdgeTriggeredEL2(target_el, syndrome) then, exception, preferred_exception_return, vect_offset;
    else
        assert PSTATE.EL IN {
            ClearPendingPhysicalSErrorEL0();

    exception =, ExceptionSyndromeEL1();Exception_SErrorAArch64.TakeException);
    exception.syndrome = syndrome;{
    AArch64.TakeExceptionEL1(target_el, exception, preferred_exception_return, vect_offset);, exception,
```

## Library pseudocode for aarch64/exceptions/

### async/AArch64.TakeVirtualFIQExceptionAArch64.TakePhysicalSErrorException

```
// AArch64.TakeVirtualFIQException()
// =====// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {AArch64.TakePhysicalSErrorException(bits(25) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled());
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = () &&
        (HCR_EL2.TGE == '1' || (! IsInHost() && HCR_EL2.AMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    vect_offset = 0x180;

    exception = bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, syndrome) then
        ClearPendingPhysicalSError();

    exception = ExceptionSyndrome(Exception_FIQException_SError);
    exception.syndrome = syndrome;

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);(target_el, exception,
```

## Library pseudocode for aarch64/exceptions/

### async/AArch64.TakeVirtualIRQExceptionAArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualIRQException()
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualIRQException()
AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_IRQException_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/

### async/AArch64.TakeVirtualSErrorExceptionAArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualSErrorException()
// =====// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualSErrorException(bits(25) syndrome)

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_SErrorException_IRQ);

    if HaveRASExt() then
        exception.syndrome<24> = VESR_EL2.IDS;
        exception.syndrome<23:0> = VESR_EL2.ISS;
    else
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then exception.syndrome = syndrome;

    ClearPendingVirtualSError();};
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/

### exceptions/debugasync/AArch64.BreakpointExceptionAArch64.TakeVirtualSErrorException

```
// AArch64.BreakpointException()
// =====// AArch64.TakeVirtualSErrorException()
// =====

AArch64.BreakpointException(AArch64.TakeVirtualSErrorException(bits(25) syndrome))

    assert PSTATE.EL IN {FaultRecord fault})
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    ());
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    vect_offset = 0x180;
    exception = AArch64.AbortSyndromeExceptionSyndrome(Exception_BreakpointException_SError, fault, vaddr
    );

    if PSTATE.EL == if EL2HaveRASExt || route_to_el2 then() then
        exception.syndrome<24> = VESR_EL2.IDS;
        exception.syndrome<23:0> = VESR_EL2.ISS;
    else
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then exception.syndrome = syndrome;
        AArch64.TakeExceptionClearPendingVirtualSError(EL2, exception, preferred_exception_return, vect_o
    else();
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

**Library pseudocode for aarch64/exceptions/  
debug/AArch64.SoftwareBreakpointAArch64.BreakpointException**

```
// AArch64.SoftwareBreakpoint()
// =====// AArch64.BreakpointException()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

route_to_el2 = (PSTATE.EL IN {AArch64.BreakpointException(FaultRecord fault)
assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

vaddress = bits(64) UNKNOWN;
exception = ExceptionSyndromeAArch64.AbortSyndrome(Exception_SoftwareBreakpointException_Breakpoint),
exception.syndrome<15:0> = immediate;
, fault, vaddress);

if if PSTATE.EL == UInt(EL2(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elseif route_to_el2 then|| route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

**Library pseudocode for aarch64/exceptions/  
debug/AArch64.SoftwareStepExceptionAArch64.SoftwareBreakpoint**

```
// AArch64.SoftwareStepException()
// =====// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareStepException()
assert PSTATE.EL !=AArch64.SoftwareBreakpoint(bits(16) immediate)

route_to_el2 = (PSTATE.EL IN { EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareStepException_SoftwareBreakpoint);
exception.syndrome<15:0> = immediate;

if SoftwareStep_DidNotStepUInt() then
    exception.syndrome<24> = '0';
else
    exception.syndrome<24> = '1';
    exception.syndrome<6> = if(PSTATE.EL) > SoftwareStep_SteppedEXUInt() then '1' else '0';
    exception.syndrome<5:0> = '100010'; // IFSC = Debug Exception

if PSTATE.EL == ( ) then
    AArch64.TakeExceptionEL2EL1 || route_to_el2 then(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elseif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ debug/AArch64.VectorCatchExceptionAArch64.SoftwareStepException

```
// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.// AArch64.SoftwareStepException()
// =====

AArch64.VectorCatchException(AArch64.SoftwareStepException(
    assert PSTATE.EL != FaultRecordEL3 fault)
    assert PSTATE.EL !=;

    route_to_el2 = (PSTATE.EL IN { EL2EL0;
    assert, EL1} && EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    () &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndromeExceptionSyndrome(Exception_VectorCatchException_SoftwareStep, fault,
    if

        SoftwareStep_DidNotStep() then
            exception.syndrome<24> = '0';
        else
            exception.syndrome<24> = '1';
            exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
            exception.syndrome<5:0> = '100010'; // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ debug/AArch64.WatchpointExceptionAArch64.VectorCatchException

```
// AArch64.WatchpointException()
// =====// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.WatchpointException(bits(64) vaddress, AArch64.VectorCatchException( FaultRecord fault)
    assert PSTATE.EL != EL3EL2;

    route_to_el2 = (PSTATE.EL IN { assertEL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    () && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0; vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception =

    ExceptionRecord exception;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2WatchpointException_VectorCatch, fault, vaddress);
    else
        exception =, fault, vaddress); AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/

exceptions/~~exceptions~~debug/AArch64.ExceptionClassAArch64.WatchpointException

```

// AArch64.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit)// AArch64.WatchpointException()
// ===== AArch64.ExceptionClass(AArch64.WatchpointException(bits(64) vaddress, Ex

    il_is_valid = TRUE;
    from_32 = fault;
    assert PSTATE.EL != UsingAArch32EL3();
    integer ec;
    case exceptype of
        when;

    route_to_el2 = (PSTATE.EL IN { Exception_UncategorizedEL0          ec = 0x00; il_is_valid = FALSE;
        when, Exception_WFxTrapEL1          ec = 0x01;
        when} && Exception_CP15RRTTrapEL2Enabled          ec = 0x03; assert from_32;
        when() &&
            (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = Exception_CP15RRTTrapThisInstrAddr          ec = 0x04; assert 1
        when();
    vect_offset = 0x0; Exception_CP14RRTTrapExceptionRecord          ec = 0x05; assert from_32;
    whenexception;
    if Exception_CP14DTTTrapHaveNV2Ext          ec = 0x06; assert from_32;
        when() && fault.acctype == Exception_AdvSIMDFPAccessTrapAccType_NV2REGISTER          ec = 0x07;
        whenthen
            exception = Exception_FPIDTrapAArch64.AbortSyndrome          ec = 0x08;
        when{ Exception_PACTrapException_NV2Watchpoint          ec = 0x09;
        when, fault, vaddress);
    else
        exception = Exception_LDST64BTrapAArch64.AbortSyndrome          ec = 0x0A;
        when{ Exception_CP14RRTTrap          ec = 0x0C; assert from_32;
        when Exception_BranchTarget          ec = 0x0D;
        when Exception_IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall          ec = 0x11;
        when Exception_HypervisorCall          ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap          ec = 0x18; assert !from_32;
        when Exception_ERetTrap          ec = 0x1A; assert !from_32;
        when Exception_PACFail          ec = 0x1C; assert !from_32;
        when Exception_InstructionAbort          ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment          ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort          ec = 0x24;
        when Exception_NV2DataAbort          ec = 0x25;
        when Exception_SPAlignment          ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_MemCpyMemSet          ec = 0x27;
        when Exception_FPtrappedException          ec = 0x28;
        when Exception_SEError          ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint          ec = 0x30; il_is_valid = FALSE;
        when Exception_SoftwareStep          ec = 0x32; il_is_valid = FALSE;
        when Exception_Watchpoint          ec = 0x34; il_is_valid = FALSE;
        when, fault, vaddress);

    if PSTATE.EL == Exception_NV2WatchpointEL2          ec = 0x35; il_is_valid = FALSE;
        when|| route_to_el2 then Exception_SoftwareBreakpointAArch64.TakeException          ec = 0x38;
        when{ Exception_VectorCatchEL2          ec = 0x3A; il_is_valid = FALSE; assert from_32;
        otherwise, exception, preferred_exception_return, vect_offset);
    else
        UnreachableAArch64.TakeException();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;
    bit il;
    if il_is_valid then
        il = if{ ThisInstrLengthEL1() == 32 then '1' else '0';
    else
        il = '1';

```

```
assert from_32 || il == '1'; // AArch64 instructions always 32-bit
```

```
return (ec,il);, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/  
exceptions/**AArch64.ReportException****AArch64.ExceptionClass**

```

// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.// AArch64.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit)

AArch64.ReportException(AArch64.ExceptionClass(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;
exceptype, bits(2) target_el)

    (ec,il) = il_is_valid = TRUE;
    from_32 = AArch64.ExceptionClassUsingAArch32(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';();
    integer ec;
    case exceptype of
        when

            ESRException_Uncategorized[target_el] = (ec = 0x00; il_is_valid = FALSE;
            when ZerosException_WFxFTrap(27) : // <63:37>
                iss2 : // <36:32>
                ec<5:0> : // <31:26>
                il : // <25>
                iss); // <24:0>

    if exceptype IN {ec = 0x01;
        when
            Exception_CP15RTTTrap ec = 0x03; assert from_32;
            when Exception_CP15RRTTTrap ec = 0x04; assert from_32;
            when Exception_CP14RTTTrap ec = 0x05; assert from_32;
            when Exception_CP14DTTTrap ec = 0x06; assert from_32;
            when Exception_AdvSIMDFPAccessTrap ec = 0x07;
            when Exception_FPIDTrap ec = 0x08;
            when Exception_PACTrap ec = 0x09;
            when Exception_LDST64BTrap ec = 0x0A;
            when Exception_CP14RRTTTrap ec = 0x0C; assert from_32;
            when Exception_BranchTarget ec = 0x0D;
            when Exception_IllegalState ec = 0x0E; il_is_valid = FALSE;
            when Exception_SupervisorCall ec = 0x11;
            when Exception_HypervisorCall ec = 0x12;
            when Exception_MonitorCall ec = 0x13;
            when Exception_SystemRegisterTrap ec = 0x18; assert !from_32;
            when Exception_ERetTrap ec = 0x1A; assert !from_32;
            when Exception_PACFail ec = 0x1C; assert !from_32;
            when Exception_InstructionAbort,ec = 0x20; il_is_valid = FALSE;
            when
                Exception_PCAlignment,ec = 0x22; il_is_valid = FALSE;
            when
                Exception_DataAbort,ec = 0x24;
            when
                Exception_NV2DataAbort,ec = 0x25;
            when
                Exception_NV2WatchpointException_SPAlignment,ec = 0x26; il_is_valid = FALSE; assert !from_32;
            when
                Exception_MemCpyMemSet ec = 0x27;
            when Exception_FPtrappedException ec = 0x28;
            when Exception_SError ec = 0x2F; il_is_valid = FALSE;
            when Exception_Breakpoint ec = 0x30; il_is_valid = FALSE;
            when Exception_SoftwareStep ec = 0x32; il_is_valid = FALSE;
            when Exception_Watchpoint
        } then ec = 0x34; il_is_valid = FALSE;
        when
            FARException_NV2Watchpoint[target_el] = exception.vaddress;

```

```

else ec = 0x35; il_is_valid = FALSE;
when
    FARException_SoftwareBreakpoint[target_el] = bits(64) UNKNOWN;

if exception.ipavalid then
    HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
    if ec = 0x38;
    when IsSecureEL2EnabledException_VectorCatch() && ec = 0x3A; il_is_valid = FALSE; assert from_32;
    otherwise CurrentSecurityStateUnreachable() ==();

if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
    ec = ec + 1;

if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
    ec = ec + 4;
bit il;
if il_is_valid then
    il = if SS_SecureThisInstrLength then
        HPFAR_EL2.NS = exception.NS;
    else
        HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<43:4> = bits(40) UNKNOWN;
    () == 32 then '1' else '0';
else
    il = '1';
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

return; return (ec,il);

```

## Library pseudocode for aarch64/exceptions/ exceptions/AArch64.ResetControlRegistersAArch64.ReportException

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.
AArch64.ResetControlRegisters(boolean cold_reset);AArch64.ReportException(ExceptionRecord exception, bits

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = (Zeros(27) : // <63:37>
                     iss2      : // <36:32>
                     ec<5:0>   : // <31:26>
                     il        : // <25>
                     iss);      // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_Watchpoint
    } then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if exception.ipavalid then
        HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
        if IsSecureEL2Enabled() && IsSecure() then
            HPFAR_EL2.NS = exception.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/exceptions/ exceptions/**AArch64.TakeReset****AArch64.ResetControlRegisters**

```
// AArch64.TakeReset()
// =====
// Reset into AArch64 state // Resets System registers and memory-mapped control registers that have arch
// reset values to those values.

AArch64.TakeReset(boolean cold_reset)
    assert(AArch64.ResetControlRegisters(boolean cold_reset); HaveAArch64());

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1'; // Select stack pointer
    PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0'; // Clear software step bit
    PSTATE.DIT = '0'; // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:AArch64.PAMax()>) && IsZero(rv<1:0>);

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch64/

exceptions/ieeefpexceptions/AArch64.FPTrappedExceptionAArch64.TakeReset

```

// AArch64.FPTrappedException()
// =====// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    exception = AArch64.TakeReset(boolean cold_reset);
    assert ExceptionSyndromeHaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if Exception_FPTrappedExceptionHaveEL();
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1'; // TFV
        else
            exception.syndrome<23> = '0'; // TFV
    else
        exception.syndrome<23> = '1'; // TFV
        exception.syndrome<10:8> = bits(3) UNKNOWN; // VECITR
        if exception.syndrome<23> == '1' then
            exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOf
        else
            exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = ( EL2EnabledEL3() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return =) then
        PSTATE.EL = ThisInstrAddrEL3();
    vect_offset = 0x0;

    if;
    elsif UIntHaveEL(PSTATE.EL) > ( UIntEL2() then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1) then;

    // Reset System registers and other system components
    AArch64.TakeExceptionAArch64.ResetControlRegisters(PSTATE.EL, exception, preferred_exception_return);
    elsif route_to_el2 then(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1'; // Select stack pointer
    PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0'; // Clear software step bit
    PSTATE.DIT = '0'; // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.TakeExceptionAArch64.ResetGeneralRegisters(); AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv; // IMPLEMENTATION_DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2, exception, preferred_exception_return, vect_offset);
    else) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert
        AArch64.TakeExceptionIsZero(({rv<63:()>} && IsZero(rv<1:0>));

```

```

boolean branch_conditional = FALSE;
BranchTo(rv, BranchType_RESETEL1AArch64.PAMax, exception, preferred_exception_return, vect_offset);

```

## Library pseudocode for aarch64/

### exceptions/syscallsieefp/AArch64.CallHypervisorAArch64.FPTrappedException

```

// AArch64.CallHypervisor()
// =====
// Performs a HVC call// AArch64.FPTrappedException()
// =====

AArch64.CallHypervisor(bits(16) immediate)
    assertAArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    exception = HaveELExceptionSyndrome(EL2Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1'; // TFV
        else
            exception.syndrome<23> = '0'; // TFV
    else
        exception.syndrome<23> = '1'; // TFV
        exception.syndrome<10:8> = bits(3) UNKNOWN; // VECITR
        if exception.syndrome<23> == '1' then
            exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
        else
            exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    if route_to_el2 = UsingAArch32EL2Enabled() then() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = AArch32.ITAdvanceThisInstrAddr();();
    vect_offset = 0x0;

    if
        SSAdvanceUInt();
        bits(64) preferred_exception_return = (PSTATE.EL) -> NextInstrAddrUInt();
        vect_offset = 0x0;

    exception = ( ExceptionSyndromeEL1() ) then Exception_HypervisorCallAArch64.TakeException);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == (PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then EL3 then
        AArch64.TakeException(EL3EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2EL1, exception, preferred_exception_return, vect_offset);

```

## Library pseudocode for aarch64/exceptions/ syscalls/AArch64.CallSecureMonitorAArch64.CallHypervisor

```
// AArch64.CallSecureMonitor()
// =====// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallSecureMonitor(bits(16) immediate)
AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL3EL2) && !;

    ifELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCallException_HypervisorCall);
    exception.syndrome<15:0> = immediate; exception.syndrome<15:0> = immediate;

    if PSTATE.EL ==
        EL3 then
            AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ syscalls/AArch64.CallSupervisorAArch64.CallSecureMonitor

```
// AArch64.CallSupervisor()
// =====
// Calls the Supervisor// AArch64.CallSecureMonitor()
// =====

AArch64.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    ifAArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == bits(64) preferred_exception_return = EL0 && EL2Enabled() && HCR_EL2.T
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCallException_MonitorCall);
    exception.syndrome<15:0> = immediate;

    if exception.syndrome<15:0> = immediate; UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeExceptionEL3(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/  
exceptions/~~takeexception~~~~syscalls~~/AArch64.TakeExceptionAArch64.CallSupervisor

```

// AArch64.TakeException()
// =====
// Take an exception to an Exception level using AArch64.// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.TakeException(bits(2) target_el, AArch64.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if ExceptionRecordUsingAArch32 exception_in,
        bits(64) preferred_exception_return, integer vect_offset_in)
    assert() then HaveELAArch32.ITAdvance(target_el) && !(); ELUsingAArch32.SSAdvance(target_el) &&();
    route_to_el2 = PSTATE.EL == UIntEL0(target_el) >= && UIntEL2Enabled(PSTATE.EL); () && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return =
        ExceptionRecordNextInstrAddr exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if();
    vect_offset = 0x0;

    exception = HaveIESBExceptionSyndrome() then
        sync_errors = ( SCTLRException_SupervisorCall[target_el].IESB == '1';
        if);
    exception.syndrome<15:0> = immediate;

    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
            iesb_req = FALSE;
            sync_errors = FALSE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
        else
            sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    integer vect_offset = vect_offset_in;
    if UInt(target_el) > (PSTATE.EL) -> UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == ( EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
            elsif) then IsInHostAArch64.TakeException() && PSTATE.EL == (PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route to el2 then EL0AArch64.TakeException && target_el == ( EL2 then
            lower_32 =, exception, preferred_exception_return, vect_offset);
        else ELUsingAArch32AArch64.TakeException(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
            vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

        elsif PSTATE.SP == '1' then
            vect_offset = vect_offset + 0x200;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState);

    if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
        if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
            spsr<3:2> = '10';
        else
            if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
                spsr<3:2> = '10';

    if HaveBTIExt() && !UsingAArch32() then

```

```

        boolean zero_btype;
        // SPSR[].BTYP is only guaranteed valid for these exception types
        if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                                   Exception_SoftwareStep, Exception_PCAlignment,
                                   Exception_InstructionAbort, Exception_Breakpoint,
                                   Exception_VectorCatch, Exception_SoftwareBreakpoint,
                                   Exception_IllegalState, Exception_BranchTarget} then
            zero_btype = FALSE;
        else
            zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYP);
        if zero_btype then spsr<11:10> = '00';

        if HaveNV2Ext() && exception.exceptype == Exception_NV2DataAbort && target_el == EL3 then
            // External aborts are configured to be taken to EL3
            exception.exceptype = Exception_DataAbort;
        if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
            AArch64.ReportException(exception, target_el);

        PSTATE.EL = target_el;
        PSTATE.nRW = '0';
        PSTATE.SP = '1';

        SPSR[] = spsr;
        ELR[] = preferred_exception_return;

        PSTATE.SS = '0';
        if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLRL[].SPINTMASK;
        PSTATE.<D,A,I,F> = '1111';
        PSTATE.IL = '0';
        if from_32 then // Coming from AArch32
            PSTATE.IT = '00000000';
            PSTATE.T = '0'; // PSTATE.J is RES0
        if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
            SCTLRL[].SPAN == '0') then
            PSTATE.PAN = '1';
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveBTIExt() then PSTATE.BTYP = '00';
        if HaveSSBSExt() then PSTATE.SSBS = SCTLRL[].DSSBS;
        if HaveMTEEExt() then PSTATE.TCO = '1';

        boolean branch_conditional = FALSE;
        BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);

        CheckExceptionCatch(TRUE); // Check for debug event on exception entry

        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

        EndOfInstruction();, exception, preferred_exception_return, vect_offset);

```

Library pseudocode for aarch64/

exceptions/~~trap~~~~takeexception~~/AArch64.AArch32SystemAccessTrapAArch64.TakeException

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AArch32 system register access.// AArch64.TakeException()
// =====
// Take an exception to an Exception level using AArch64.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assertAArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                                bits(64) preferred_exception_return, integer vect_offset_in)
    assert HaveEL(target_el) && target_el != (target_el) && ! EL0ELUsingAArch32 && (target_el) && UInt(target_el) <= EL3

    bits(64) preferred_exception_return = (PSTATE.EL); ThisInstrAddrExceptionRecord();
    vect_offset = 0x0;

    exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if AArch64.AArch32SystemAccessTrapSyndromeHaveIESB() then
        sync_errors = ThisInstrSCTLR(), ec); [target_el].IESB == '1';
        if
        () then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
            iesb_req = FALSE;
            sync_errors = FALSE;
            TakeUnmaskedPhysicalErrorInterrupts(iesb_req);
    else
        sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    integer vect_offset = vect_offset_in;
    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
            lower_32 = ELUsingAArch32(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState);

    if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
        if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
            spsr<3:2> = '10';
        else
            if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
                spsr<3:2> = '10';

    if HaveBTIExt() && !UsingAArch32() then
        boolean zero_btype;
        // SPSR[].BTYP is only guaranteed valid for these exception types
        if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                                Exception_SoftwareStep, Exception_PCAlignment,
                                Exception_InstructionAbort, Exception_Breakpoint,
                                Exception_VectorCatch, Exception_SoftwareBreakpoint,
                                Exception_IllegalState, Exception_BranchTarget} then

```

```

        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';

    if HaveNV2Ext() && exception.exceptype == Exception_NV2DataAbort && target_el == EL3 then
        // External aborts are configured to be taken to EL3
        exception.exceptype = Exception_DataAbort;
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = spsr;
    ELR[] = preferred_exception_return;

    PSTATE.SS = '0';
    if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLR[].SPINTMASK;
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAQExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
    if HaveMTEExt() then PSTATE.TCO = '1';

    boolean branch_conditional = FALSE;
    BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstructionAArch64.TakeExceptionHaveDoubleFaultExt(target_el, exception, preferred_exception_ret

```

Library pseudocode for aarch64/exceptions/

traps/~~AArch64.AArch32SystemAccessTrapSyndrome~~AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AArch32 system register access. AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer
assert
ExceptionRecordHaveEL exception;

case ec of
    when 0x0 exception = (target_el) && target_el != ExceptionSyndromeEL0(&&Exception_Uncategorized);
    when 0x3 exception = (target_el) >= ExceptionSyndromeUInt((PSTATE.EL);

bits(64) preferred_exception_return = Exception_CP15RTTTrapThisInstrAddr;
    when 0x4 exception = ();
vect_offset = 0x0;

exception = ExceptionSyndromeAArch64.AArch32SystemAccessTrapSyndrome(Exception_CP15RRTTTrapThisInstr);
    when 0x5 exception = (), ec); ExceptionSyndromeAArch64.TakeException(Exception_CP14RTTTrap);
    when 0x6 exception = ExceptionSyndrome(Exception_CP14DTTTrap);
    when 0x7 exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
    when 0x8 exception = ExceptionSyndrome(Exception_FPIDTrap);
    when 0xC exception = ExceptionSyndrome(Exception_CP14RRTTTrap);
    otherwise Unreachable();

bits(20) iss = Zeros();

if exception.exceptype == Exception_Uncategorized then
    return exception;
elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTTrap, Exception_CP15RTTTrap} then
    // Trapped MRC/MCR, VMRS on FPSID
    if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
        iss<19:17> = instr<7:5>; // opc2
        iss<16:14> = instr<23:21>; // opc1
        iss<13:10> = instr<19:16>; // CRn
        iss<4:1> = instr<3:0>; // CRm
    else
        iss<19:17> = '000';
        iss<16:14> = '111';
        iss<13:10> = instr<19:16>; // reg
        iss<4:1> = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookupRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elsif exception.exceptype IN {Exception_CP14RRTTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>; // opc1
        if instr<19:16> == '1111' then // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookupRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookupRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1> = instr<3:0>; // CRm
    elsif exception.exceptype == Exception_CP14DTTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>; // imm8
        iss<4> = instr<23>; // U
        iss<2:1> = instr<24,21>; // P,W
        if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;

```

```
    iss<3>    = '1';  
    iss<0> = instr<20>; // Direction  
  
    exception.syndrome<24:20> = ConditionSyndrome();  
    exception.syndrome<19:0> = iss;  
  
    return exception; (target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/  
traps/**AArch64.AdvSIMDFPAccessTrap****AArch64.AArch32SystemAccessTrapSyndrome**

```

// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[] // AArch64.AArch32SystemAccessTrapSyndr
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer
    ExceptionRecord exception;
    vect_offset = 0x0;

    route_to_el2 = (target_el == case ec of
        when 0x0 exception = EL1ExceptionSyndrome && ( EL2EnabledExceptionUncategorized() && HCR_EL2.T
    if route_to_el2 then
        exception =);
        when 0x3 exception = ExceptionSyndrome(ExceptionUncategorizedException_CP15RRTTrap););
        when 0x4 exception =
            AArch64.TakeExceptionExceptionSyndrome(EL2Exception_CP15RRTTrap, exception, preferred_exception_r
    else
        exception =);
        when 0x5 exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        when 0x6 exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7 exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = when 0x8 exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype == ExceptionUncategorized then
        return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RRTTrap, Exception_CP15RRTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>; // opc2
            iss<16:14> = instr<23:21>; // opc1
            iss<13:10> = instr<19:16>; // CRn
            iss<4:1> = instr<3:0>; // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>; // reg
            iss<4:1> = '0000';

            if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
                iss<9:5> = '11111';
            elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
                iss<9:5> = bits(5) UNKNOWN;
            else
                iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap}
            // Trapped MRRC/MCRR, VMRS/VMSR
            iss<19:16> = instr<7:4>; // opc1
            if instr<19:16> == '1111' then // Rt2==15
                iss<14:10> = bits(5) UNKNOWN;
            else
                iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

            if instr<15:12> == '1111' then // Rt==15
                iss<9:5> = bits(5) UNKNOWN;
            else
                iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
            iss<4:1> = instr<3:0>; // CRm
        elsif exception.exceptype == Exception_CP14DTTTrap then
            // Trapped LDC/STC
            iss<19:12> = instr<7:0>; // imm8

```

```

    iss<4>      = instr<23>;          // U
    iss<2:1>    = instr<24,21>;      // P,W
    if instr<19:16> == '1111' then  // Rn==15, LDC(Literal addressing)/STC
        iss<9:5> = bits(5) UNKNOWN;
        iss<3>   = '1';
    iss<0> = instr<20>;          // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
());
    exception.syndrome<19:0> = iss;

    return;    return exception;

```

## Library pseudocode for aarch64/exceptions/

### traps/AArch64.CheckCP15InstrCoarseTrapsAArch64.AdvSIMDFPAccessTrap

```

// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLRL_ELx. // AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
                        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLRL_EL1.TIDCP.
    if (AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
        bits(64) preferred_exception_return = HaveFeatTIDCP1ThisInstrAddr() && PSTATE.EL == (); EL0ExceptionRec
        vect_offset = 0x0;

    route_to_el2 = (target_el == IsInHost() &&
                    !ELUsingAArch32(EL1) && SCTLRL_EL1.TIDCP == '1' && trapped_encoding) then
        if && EL2Enabled() && HCR_EL2.TGE == '1' then() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception =
            AArch64.AArch32SystemAccessTrapExceptionSyndrome(EL2ExceptionUncategorized, 0x3);
        else);
            AArch64.AArch32SystemAccessTrapAArch64.TakeException(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLRL_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
            SCTLRL_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

    major = if nreg == 1 then CRn else CRm;
    // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
    // and MRC and MCR disabled by HCR_EL2.TIDCP.
    if (!, exception, preferred_exception_return, vect_offset);
    else
        exception = IsInHostExceptionSyndrome() && !(major IN {4,14}) && HSTR_EL2<major> == '1') ||
            (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == ( EL0Exception_AdvSIMDFPAccessTrap &&
                boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at);
            exception.syndrome<24:20> = EL0ConditionSyndrome") then
                UNDEFINED;();
            AArch64.AArch32SystemAccessTrapAArch64.TakeException(EL2, 0x3);(target_el, exception, prefer

    return;

```

## Library pseudocode for aarch64/exceptions/

### traps/AArch64.CheckFPAdvSIMDEnabledAArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckFPAdvSIMDEnabled()
// =====// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTL_ELx.

AArch64.CheckFPAdvSIMDEnabled()AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
                        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTL_EL1.TIDCP.
    if (
        () && PSTATE.EL == EL0 && !IsInHost() &&
        !ELUsingAArch32(EL1) && SCTL_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTL_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
            SCTL_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

    major = if nreg == 1 then CRn else CRm;
    // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
    // and MRC and MCR disabled by HCR_EL2.TIDCP.
    if ((!IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1') ||
        (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
        if (PSTATE.EL == EL0 &&
            boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
            UNDEFINED;
        AArch64.AArch32SystemAccessTrap(EL2AArch64.CheckFPEEnabledHaveFeatTIDCP1();, 0x3);
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.CheckFPAdvSIMDTrapAArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.// AArch64.CheckFPAdvSIMDEnabled()
// =====

AArch64.CheckFPAdvSIMDTrap()
    if AArch64.CheckFPAdvSIMDEnabled() HaveELAArch64.CheckFPEnabled(EL3) && CPTR_EL3.TFP == '1' && EL3SDD
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);();
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.CheckFPEnabledAArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPEnabled()
// =====
// Check against CPACR[]// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPEnabled()
AArch64.CheckFPAdvSIMDTrap()
    if PSTATE.EL IN {EL0, EL1} && !IsInHostEL2() then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == } && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '11' disabled = FALSE;
            if disabled then && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL1EL2););
        else
            if CPTR_EL2.TFP == '1' then

                (EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3AArch64.CheckFPAdvSIMDTrapAArch64.AdvSIMDFPAccessTrap(EL3));

    return;
```

## Library pseudocode for aarch64/exceptions/

### traps/AArch64.CheckForERetTrapAArch64.CheckFPEnabled

```
// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETA, ERETB instruction// AArch64.CheckFPEnabled()
// =====
// Check against CPACR[]

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETA, ERETB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
    // is trapped to EL2
    route_to_el2 = (PSTATE.EL == AArch64.CheckFPEnabled()
    if PSTATE.EL IN { EL0, EL1 } && ! EL2EnabledIsInHost() &&
        (() then
            // Check if access disabled in CPACR_EL1
            boolean disabled;
            case CPACR_EL1.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == HaveNVExtEL0() && HCR_EL2.NV == '1' ||
                (
                when '11' disabled = FALSE;
            if disabled then HaveFGTExtAArch64.AdvSIMDFPAccessTrap() && (!(HaveELEL1()); EL3AArch64.CheckFPAdvS
                HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0'; // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then // ERETA
                exception.syndrome<0> = '0';
            else // ERETB
                exception.syndrome<0> = '1';
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);();
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.CheckForSMCUndefOrTrapAArch64.CheckForERetTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)
    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
    // is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') && (
        (
            HaveELHaveNVExt() && HCR_EL2.NV == '1') ||
            (EL3HaveFGTExt) && SCR_EL3.SMD == '1') then
        UNDEFINED;
        route_to_el2 = FALSE;
        if !() && HCR_EL2.<E2H, TGE> != '11' &&
            (!HaveEL(EL3) then
                if PSTATE.EL ==) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ERET == '1')));
        if route_to_el2 then EL1ExceptionRecord && exception;
        bits(64) preferred_exception_return = EL2Enabled() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
            else
                UNDEFINED;
        else
            route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
        if route_to_el2 then
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;
            exception = ExceptionSyndrome(Exception_MonitorCallException_ERetTrap);
            exception.syndrome<15:0> = imm;
            exception.trappedsyscallinst = TRUE; if !eret_with_pac then
                exception.syndrome<1> = '0';
                exception.syndrome<0> = '0'; // RES0
            else
                exception.syndrome<1> = '1';
                if pac uses key a then // ERETAA
                    exception.syndrome<0> = '0';
                else // ERETAB
                    exception.syndrome<0> = '1';
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.CheckForSVCTrapAArch64.CheckForSMCUnDefOrTrap

```
// AArch64.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction// AArch64.CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if AArch64.CheckForSMCUnDefOrTrap(bits(16) imm)
    if PSTATE.EL == HaveFGTExt() then
        route_to_el2 = FALSE;
    if PSTATE.EL == EL0 then
        route_to_el2 = (!then UNDEFINED;
    if (!(PSTATE.EL == UsingAArch32() && !ELUsingAArch32(EL1) &&&
        EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
        (HCR_EL2.<E2H, TGE> != '11' && (!() && HCR_EL2.TSC == '1') && HaveEL(EL3) || SC

    elsif PSTATE.EL ==) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if ! HaveEL(EL3) then
    if PSTATE.EL == EL1 then
        route_to_el2 = (&&EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
        (!() then
        if HaveELHaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
            route_to_el2 = TRUE;
        else
            UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL3EL1) || SCR_EL3.FGTEn == '1'));

    if route_to_el2 then
        exception = && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_SupervisorCallException_MonitorCall);
        exception.syndrome<15:0> = immediate;
        exception.trappedsyscallinst = TRUE;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;});
        exception.syndrome<15:0> = imm;
        exception.trappedsyscallinst = TRUE;

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.CheckForWfxTrapAArch64.CheckForSVCTrap

```
// AArch64.CheckForWfxTrap()
// AArch64.CheckForSVCTrap()
// =====
// Check for trap on WFE or WFI instruction// Check for trap on SVC instruction

AArch64.CheckForWfxTrap(bits(2) target_el, AArch64.CheckForSVCTrap(bits(16) immediate)
    if WfxTypeHaveFGTExt wfxtype)
    assert() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL0) && !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
                (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(target_el)));

        boolean is_wfe = wfxtype IN {(WfxType_WFEEL3,) || SCR_EL3.FGTEn == '1'});

        elsif PSTATE.EL == WfxType_WFTEL1};
        boolean trap;
        case target_el of
            when then
                route_to_el2 = (! ELUsingAArch32(EL1
                trap = (if is_wfe then) && SCTLREL2Enabled[].nTWE else() && HFGITR_EL2.SVC_EL1 == '1' &&
                    (HCR_EL2.<E2H, TGE> != '11' && (! SCTLRHaveEL[].nTWI) == '0');

                when( EL2
                    trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
                when EL3
                    trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
            ) || SCR_EL3.FGTEn == '1')));

        if trap then if route_to_el2 then
            exception =
                (Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            exception.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

        AArch64.TakeException(EL2AArch64.WfxTrapExceptionSyndrome(wfxtype, target_el);, exception, p
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.CheckIllegalStateAArch64.CheckForWFXTrap

```
// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.// AArch64.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == AArch64.CheckForWFXTrap(bits(2) target_el, EL0WfxType &&wfxtype)
        assert EL2EnabledHaveEL() && HCR_EL2.TGE == '1';
(target_el);

        bits(64) preferred_exception_return = boolean is_wfe = wfxtype IN { ThisInstrAddrWfxType_WFE()
        vect_offset = 0x0;

        exception =, ExceptionSyndromeWfxType_WFET();
        boolean trap;
        case target_el of
            when Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then trap = (if is_wfe then
            AArch64.TakeExceptionSCTLR(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elseif route_to_el2 then [].nTWE else
            AArch64.TakeExceptionSCTLR([].nTWI) == '0';
            when EL2, exception, preferred_exception_return, vect_offset);
        elsetrap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when
            AArch64.TakeExceptionEL3(trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

        if trap then EL1AArch64.WFXTrap, exception, preferred_exception_return, vect_offset);(wfxtype, target_
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.MonitorModeTrapAArch64.CheckIllegalState

```
// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(ExceptionUncategorizedException_IllegalState);

        if IsSecureEL2EnabledUInt() then (PSTATE.EL) ->
            UInt(EL1) then
                AArch64.TakeException({PSTATE.EL, exception, preferred_exception_return, vect_offset);
            elseif route_to_el2 then AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
            else
                AArch64.TakeException(EL3EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.SystemAccessTrapAArch64.MonitorModeTrap

```
// AArch64.SystemAccessTrap()
// =====
// Trapped access to AArch64 system register or system instruction.// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assertAArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = HaveEL.ThisInstrAddr(target_el) && target_el !=();
    vect_offset = 0x0;

    exception = EL0ExceptionSyndrome &&{ UIntExceptionUncategorized(target_el) >=};

    if UIntIsSecureEL2Enabled(PSTATE.EL);

        bits(64) preferred_exception_return =() then ThisInstrAddrAArch64.TakeException();
        vect_offset = 0x0;

    exception ={ AArch64.SystemAccessTrapSyndromeEL2(, exception, preferred_exception_return, vect_offset
    AArch64.TakeExceptionEL3(target_el, exception, preferred_exception_return, vect_offset);, exception,
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.SystemAccessTrapSyndromeAArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord// AArch64.SystemAccessTrap()
// =====
// Trapped access to AArch64 system register or system instruction. AArch64.SystemAccessTrapSyndrome(bits
assert
    ExceptionRecordHaveEL exception;
    bits(32) instr = instr_in;
    case ec of
        when 0x0 // Trapped access due to unknown reason
            exception = (target_el) && target_el != ExceptionSyndromeEL0(&&ExceptionUncategorizedUInt);
        when 0x7 // Trapped access to SVE, Advance SIMD
            exception = (target_el) >= ExceptionSyndromeUInt((PSTATE.EL);

    bits(64) preferred_exception_return = Exception_AdvSIMDFPAccessTrapThisInstrAddr;
    exception.syndrome<24:20> =();
    vect_offset = 0x0;

    exception = ConditionSyndromeAArch64.SystemAccessTrapSyndrome();
    when 0x18 // Trapped access to system register
        exception ={ ExceptionSyndrome(Exception_SystemRegisterTrap);
        instr = ThisInstr();
        exception.syndrome<21:20> = instr<20:19>; // Op0
        exception.syndrome<19:17> = instr<7:5>; // Op2
        exception.syndrome<16:14> = instr<18:16>; // Op1
        exception.syndrome<13:10> = instr<15:12>; // CRn
        exception.syndrome<9:5> = instr<4:0>; // Rt
        exception.syndrome<4:1> = instr<11:8>; // CRm
        exception.syndrome<0> = instr<21>; // Direction
    otherwise(, ec);
    UnreachableAArch64.TakeException();

    return exception;(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/ traps/AArch64.UndefinedFaultAArch64.SystemAccessTrapSyndrome

```
// AArch64.UndefinedFault()
// =====// AArch64.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.
ExceptionRecord
AArch64.UndefinedFault()
    route_to_el2 = PSTATE.EL == AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec) EL0Exception
    bits(32) instr = instr_in;
    case ec of
        when 0x0 // Trapped access due to unknown reason
            exception = EL2Enabled() && HCR_EL2.TGE == '1';
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

            exception = ExceptionSyndrome(Exception_Uncategorized);

        if when 0x7 // Trapped access to SVE, AdvSIMD, or AdvSIMDFP
            exception = UIntExceptionSyndrome(PSTATE.EL) > UIntException_AdvSIMDFPAccessTrap();
            exception.syndrome<24:20> = EL1ConditionSyndrome then();
        when 0x18 // Trapped access to system register
            exception = AArch64.TakeExceptionExceptionSyndrome(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elseif route_to_el2 then
            AArch64.TakeExceptionException_SystemRegisterTrap();
            instr = EL2ThisInstr, exception, preferred_exception_return, vect_offset);
        else();
            exception.syndrome<21:20> = instr<20:19>; // Op0
            exception.syndrome<19:17> = instr<7:5>; // Op2
            exception.syndrome<16:14> = instr<18:16>; // Op1
            exception.syndrome<13:10> = instr<15:12>; // CRn
            exception.syndrome<9:5> = instr<4:0>; // Rt
            exception.syndrome<4:1> = instr<11:8>; // CRm
            exception.syndrome<0> = instr<21>; // Direction
        otherwise
            AArch64.TakeExceptionUnreachable(EL1, exception, preferred_exception_return, vect_offset);();

    return exception;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrapAArch64.UndefinedFault

```
// AArch64.WFxTrap()
// =====// AArch64.UndefinedFault()
// =====

AArch64.WFxTrap(AArch64.UndefinedFault())

route_to_el2 = PSTATE.EL == WFxTypeEL0 wfxtype, bits(2) target_el)
assert && UIntEL2Enabled(target_el) >() && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = UInt(PSTATE.EL);

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_WFxTrapException_Uncategorized);
exception.syndrome<24:20> =
if ConditionSyndromeUInt();

case wfxtype of
when (PSTATE.EL) > WFxType_WFIUInt
exception.syndrome<1:0> = '00';
when ( WFxType_WFE
exception.syndrome<1:0> = '01';
when WFxType_WFIT
exception.syndrome<1:0> = '10';
if HaveFeatWFXT2() then
exception.syndrome<2> = '1'; // Register field is valid
exception.syndrome<9:5> = ThisInstr().<4:0>;
else
exception.syndrome<2> = '0'; // Register field is invalid
when WFxType_WFET
exception.syndrome<1:0> = '11';
if HaveFeatWFXT2() then
exception.syndrome<2> = '1'; // Register field is valid
exception.syndrome<9:5> = ThisInstr().<4:0>;
else
exception.syndrome<2> = '0'; // Register field is invalid

if target_el == EL1 && then EL2EnabledAArch64.TakeException() && HCR_EL2.TGE == '1' then (PSTATE.EL,
elseif route_to_el2 then
AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
AArch64.TakeException(EL1(target_el, exception, preferred_exception_return, vect_offset);, except
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64AArch64.WFxTrap

```
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper// AArch64.WFxTrap()
// =====

CheckFPAdvSIMDEnabled64()AArch64.WFxTrap(
    wfxtype, bits(2) target_el)
assert UInt(target_el) > UInt(PSTATE.EL);

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_WFxTrap);
exception.syndrome<24:20> = ConditionSyndrome();

case wfxtype of
    when WFxType_WFI
        exception.syndrome<1:0> = '00';
    when WFxType_WFE
        exception.syndrome<1:0> = '01';
    when WFxType_WFIT
        exception.syndrome<1:0> = '10';
        if HaveFeatWFXT2() then
            exception.syndrome<2> = '1'; // Register field is valid
            exception.syndrome<9:5> = ThisInstr()<4:0>;
        else
            exception.syndrome<2> = '0'; // Register field is invalid
    when WFxType_WFET
        exception.syndrome<1:0> = '11';
        if HaveFeatWFXT2() then
            exception.syndrome<2> = '1'; // Register field is valid
            exception.syndrome<9:5> = ThisInstr()<4:0>;
        else
            exception.syndrome<2> = '0'; // Register field is invalid

if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeExceptionAArch64.CheckFPAdvSIMDEnabledWfxType();(target_el, exception, preferred_exce
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPEEnabled64CheckFPAdvSIMDEnabled64

```
// CheckFPEEnabled64()
// =====
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPEEnabled64()CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPEEnabledAArch64.CheckFPAdvSIMDEnabled();
```

```

// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions// CheckFPEnabled64()
// =====
// AArch64 instruction wrapper

CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = CheckFPEnabled64() ZeroExtendAArch64.CheckFPEnabled('10'); // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
    else
        target_el = EL1;

    if (!trap && EL2Enabled() && HaveFeatHGX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);();

```

```

// CheckST64BV0Enabled()
// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64BV0 instruction// Checks for trap on ST64B and LD64B instructions

CheckST64BV0Enabled()
CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('1'); // 0x1
    ('10'); // 0x2
    bits(2) target_el;

    if (PSTATE.EL != if PSTATE.EL == EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SDDTrapPriority()) then
        UNDEFINED;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;

    if (!trap && else
        target_el = EL1;

    if (!trap && EL2Enabled() && HaveFeatHCRX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
        () || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then
        if target_el == EL3 && Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else;

    if trap then
        LDST64BTrap(target_el, iss);

```

```

// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64BV0 instruction

CheckST64BVEnabled()
CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZerosZeroExtend();
    ('1'); // 0x1
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        () || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then LDST64BTrap(target_el, iss);

```

## Library pseudocode for aarch64/exceptions/traps/LDST64BTrapCheckST64BVEnabled

```
// LDST64BTrap()
// =====
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

LDST64BTrap(bits(2) target_el, bits(25) iss)
    bits(64) preferred_exception_return = CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = ThisInstrAddrZeros();
    vect_offset = 0x0;
    bits(2) target_el;

    exception = if PSTATE.EL == ExceptionSyndromeEL0() then
        if !Exception_LDST64BTrapIsInHost();
    exception.syndrome = iss;() then
        trap = SCTLR_EL1.EnASR == '0';
        target_el = if
            () && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrapAArch64.TakeExceptionEL2Enabled(target_el, exception, preferred_exception_return)

    return;(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/WFETrapDelayLDST64BTrap

```
// WFETrapDelay()
// =====
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer)// LDST64BTrap()
// =====
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when LDST64BTrap(bits(2) target_el, bits(25) iss)
            bits(64) preferred_exception_return = EL1ThisInstrAddr
            if !();
            vect_offset = 0x0;

    exception = IsInHostExceptionSyndrome() then
        delay_enabled = SCTLR_EL1.TWEDEn == '1';
        delay = 1 << (UIntException_LDST64BTrap(SCTLR_EL1.TWEDEL) + 8);
    else
        delay_enabled = SCTLR_EL2.TWEDEn == '1';
        delay = 1 << ();
    exception.syndrome = iss; UIntAArch64.TakeException(SCTLR_EL2.TWEDEL) + 8);
    when EL2
        assert EL2Enabled();
        delay_enabled = HCR_EL2.TWEDEn == '1';
        delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
    when EL3
        delay_enabled = SCR_EL3.TWEDEn == '1';
        delay = 1 << (UInt(SCR_EL3.TWEDEL) + 8);
(target_el, exception, preferred_exception_return, vect_offset);

    return (delay_enabled, delay); return;
```

## Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelayWFETrapDelay

```
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// WFETrapDelay()
// =====
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.
boolean
(boolean, integer) WaitForEventUntilDelay(boolean delay_enabled, integer delay); WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTL_EL1.TWEDEn == '1';
                delay = 1 << (UInt(SCTL_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTL_EL2.TWEDEn == '1';
                delay = 1 << (UInt(SCTL_EL2.TWEDEL) + 8);
        when EL2
            assert EL2Enabled();
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

    return (delay_enabled, delay);
```

## Library pseudocode for

aarch64/functionexceptions/aborttraps/AArch64.FaultSyndromeWaitForEventUntilDelay

```
// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.
(bits(25), bits(5))// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.
boolean AArch64.FaultSyndrome(boolean d_side, WaitForEventUntilDelay(boolean delay_enabled, integer delay)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros();
    bits(5) iss2 = Zeros();

    if !HaveFeatLS64() && HaveRASExt() && IsAsyncAbort(fault) then
        iss<12:11> = fault.errortype; // SET

    if d_side then
        if HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 then
            if (fault.statuscode IN {Fault_AccessFlag,
                Fault_Translation, Fault_Permission}) then
                (iss2, iss<24:14>, iss<12:11>) = LS64InstructionSyndrome();
            else
                if (IsSecondStage(fault) && !fault.s2fslwalk &&
                    (!IsExternalSyncAbort(fault) ||
                    (!HaveRASExt() && fault.acctype == AccType_TTW &&
                    boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
                    iss<24:14> = LSInstructionSyndrome();

        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1'; // Fault is generated by use of VNCR_EL2

        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fslwalk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);
```

## Library pseudocode for aarch64/functions/ aborts/LS64InstructionSyndromeAArch64.FaultSyndrome

```
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field, and LST.
(bits(5), bits(11), bits(2))// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.

(bits(25), bits(5)) LS64InstructionSyndrome(); AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros();
    bits(5) iss2 = Zeros();

    if !HaveFeatLS64() && HaveRASExt() && IsAsyncAbort(fault) then
        iss<12:11> = fault.errortype; // SET

    if d_side then
        if HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 then
            if (fault.statuscode IN {Fault_AccessFlag,
                Fault_Translation, Fault_Permission}) then
                (iss2, iss<24:14>, iss<12:11>) = LS64InstructionSyndrome();
            else
                if (IsSecondStage(fault) && !fault.s2fslwalk &&
                    (!IsExternalSyncAbort(fault) ||
                     (!HaveRASExt() && fault.acctype == AccType_TTW &&
                      boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
                    iss<24:14> = LSInstructionSyndrome();

                if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
                    iss<13> = '1'; // Fault is generated by use of VNCR_EL2

                if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
                    iss<8> = '1'; iss<6> = '1';
                else
                    iss<6> = if fault.write then '1' else '0';

                if IsExternalAbort(fault) then iss<9> = fault.extflag;
                iss<7> = if fault.s2fslwalk then '1' else '0';
                iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);
```

## Library pseudocode for aarch64/

### functions/cacheaborts/AArch64.DataMemZeroLS64InstructionSyndrome

```
// AArch64.DataMemZero()
// =====
// Write Zero to data memory// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field, and LST.
// (bits(5), bits(11), bits(2))

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, LS64InstructionSyndrome(); AddressDescriptor memaddrdesc_in;
    iswrite = TRUE;
    AddressDescriptor memaddrdesc = memaddrdesc_in;
    for i = 0 to size-1
        accdesc = CreateAccessDescriptor(AccType_DCZVA);
        if HaveMTEExt() then
            if AArch64.AccessIsTagChecked(vaddress, AccType_DCZVA) then
                bits(4) ptag = AArch64.PhysicalTag(vaddress);
                if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                    if boolean IMPLEMENTATION_DEFINED "DC_ZVA tag fault reported with lowest faulting address" then
                        AArch64.TagCheckFault(vaddress, AccType_DCZVA, iswrite);
                    else
                        AArch64.TagCheckFault(regval, AccType_DCZVA, iswrite);
                memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros());
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
                memaddrdesc.address.address = memaddrdesc.address.address + 1;
    return;
```

## Library pseudocode for aarch64/functions/cache/AArch64.TagMemZeroAArch64.DataMemZero

```
// AArch64.TagMemZero()
// =====
// Write Zero to tag memory// AArch64.DataMemZero()
// =====
// Write Zero to data memory

AArch64.TagMemZero(bits(64) vaddress_in, integer size)
    bits(64) vaddress = vaddress_in;
    integer count = size >> AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, LOG2_TAG_GRANULEAddressDescriptor memaddrdesc_in;
    bits(4) tag = memaddrdesc_in, integer size);
    iswrite = TRUE; AArch64.AllocationTagFromAddressAddressDescriptor(vaddress);
    for i = 0 to count-1 memaddrdesc = memaddrdesc_in;
    for i = 0 to size-1
        accdesc =
            AArch64.MemTagCreateAccessDescriptor(vaddress, { AccType_NORMAL AccType_DCZVA } = tag;
        vaddress = vaddress +);
    if () then
        if AArch64.AccessIsTagChecked(vaddress, AccType_DCZVA) then
            bits(4) ptag = AArch64.PhysicalTag(vaddress);
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                if boolean IMPLEMENTATION_DEFINED "DC_ZVA tag fault reported with lowest faulting address" then
                    AArch64.TagCheckFault(vaddress, AccType_DCZVA, iswrite);
                else
                    AArch64.TagCheckFault(regval, AccType_DCZVA, iswrite);
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros());
            if IsFault(memstatus) then
                HandleExternalWriteAbortTAG_GRANULEHaveMTEExt;
        (memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.address.address = memaddrdesc.address.address + 1;
    return;
```

## Library pseudocode for aarch64/

### functions/exclusivecache/AArch64.ExclusiveMonitorsPassAArch64.TagMemZero

```
// AArch64.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean// AArch64.TagMemZero()
// =====
// Write Zero to tag memory AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AArch64.TagMemZero(bits(64) vaddress_in, integer size)
    bits(64) vaddress = vaddress_in;
    integer count = size >> AccType_ATOMICLOG2_TAG_GRANULE;
    iswrite = TRUE;

    aligned = bits(4) tag = AArch64.CheckAlignmentAArch64.AllocationTagFromAddress(address, size, acctype);

    passed = (vaddress);
    for i = 0 to count-1 AArch64.IsExclusiveVAArch64.MemTag(address, {vaddress, ProcessorIDAcctype_NORMAL);
    if !passed then
        return FALSE;

    memaddrdesc = tag;
    vaddress = vaddress + AArch64.TranslateAddressTAG_GRANULE(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;;
return;
```

**Library pseudocode for aarch64/functions/  
exclusive/AArch64.IsExclusiveVA**

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
// AArch64.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size); AArch64.ExclusiveMonitorsPass()

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

acctype = AccType_ATOMIC;
iswrite = TRUE;

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
if !passed then
    return FALSE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
ClearExclusiveLocal(ProcessorID());

if passed then
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

return passed;
```

**Library pseudocode for aarch64/functions/  
exclusive/AArch64.MarkExclusiveVA**

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.// An optional IMPLEMENTATION DEFINED test for an exclusive access
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size); AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

**Library pseudocode for aarch64/functions/  
exclusive/AArch64.SetExclusiveMonitorsAArch64.MarkExclusiveVA**

```
// AArch64.SetExclusiveMonitors()  
// =====  
// Sets the Exclusives monitors for the current PE to record the addresses associated  
// with the virtual address region of size bytes starting at address. // Optionally record an exclusive a  
// starting at address for processorid.  
  
AArch64.SetExclusiveMonitors(bits(64) address, integer size)  
    acctype = AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size); AccType_ATOMI  
    iswrite = FALSE;  
  
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);  
  
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);  
    // Check for aborts or debug exceptions  
    if IsFault(memaddrdesc) then  
        return;  
  
    if memaddrdesc.memattrs.shareability != Shareability_NSH then  
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);  
  
    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);  
  
    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch64/

### functions/~~fusedrstpeexclusive~~/FPRSqrtStepFusedAArch64.SetExclusiveMonitors

```
// FPRSqrtStepFused()
// =====

bits(N) // AArch64.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address. FPRSqrtStepFused(bits(N) op1_in, b
assert N IN {16, 32, 64};
bits(N) result;
bits(N) op1 = op1_in;
boolean done; AArch64.SetExclusiveMonitors(bits(64) address, integer size)
acctype =
FPCRTypeAccType_ATOMIC fpcr = FPCR[];
op1 =;
iswrite = FALSE;

aligned = FPNegAArch64.CheckAlignment(op1);
boolean altfp =(address, size, acctype, iswrite);

memaddrdesc = HaveAltFPAArch64.TranslateAddress() && fpcr.AH == '1';
boolean fpexc = !altfp; // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

(type1,sign1,value1) =(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if FPUntpackIsFault(op1, fpcr, fpexc);
(type2,sign2,value2) =(memaddrdesc) then
return;

if memaddrdesc.memattrs.shareability != FPUntpackShareability_NSH(op2, fpcr, fpexc);
(done,result) =then FPProcessNaNsMarkExclusiveGlobal(type1, type2, op1, op2, fpcr, FALSE, fpexc);(mem
FPRoundingProcessorID rounding =( ), size); FPRoundingModeMarkExclusiveLocal(fpcr);

if !done then
inf1 = (type1 ==(memaddrdesc.paddress, FPType_InfinityProcessorID);
inf2 = (type2 ==( ), size); FPType_InfinityAArch64.MarkExclusiveVA);
zero1 = (type1 ==(address, FPType_ZeroProcessorID);
zero2 = (type2 == FPType_Zero);

if (inf1 && zero2) || (zero1 && inf2) then
result = FPOnePointFive('0');
elseif inf1 || inf2 then
result = FPIfinity(sign1 EOR sign2);
else
// Fully fused multiply-add and halve
result_value = (3.0 + (value1 * value2)) / 2.0;
if result_value == 0.0 then
// Sign of exact zero result depends on rounding mode
sign = if rounding == FPRounding_NEGINF then '1' else '0';
result = FPZero(sign);
else
result = FPRound(result_value, fpcr, rounding, fpexc);

return result;( ), size);
```

```

// FPRcipStepFused()
// FPRSqrtStepFused()
// =====

bits(N) FPRcipStepFused(bits(N) op1_in, bits(N) op2)
FPRSqrtStepFused(bits(N) op1_in, bits(N) op2)
  assert N IN {16, 32, 64};
  bits(N) result;
  bits(N) op1 = op1_in;
  bits(N) result;
  boolean done;
  FPCRType fpcr = FPCR[];
  op1 = FPNeg(op1);

  boolean altfp = HaveAltFP() && fpcr.AH == '1';
  boolean fpexc = !altfp;                                // Generate no floating-point exceptions
  if altfp then fpcr.<FIZ,FZ> = '11';                      // Flush denormal input and output to zero
  if altfp then fpcr.RMode = '00';                        // Use RNE rounding mode

  (type1,sign1,value1) = FPUunpack(op1, fpcr, fpexc);
  (type2,sign2,value2) = FPUunpack(op2, fpcr, fpexc);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
  FPRounding rounding = FPRoundingMode(fpcr);

  if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
      result = FPTwoFPOnePointFive('0');
    elsif inf1 || inf2 then
      result = FPIfinity(sign1 EOR sign2);
    else
      // Fully fused multiply-add
      result_value = 2.0 + (value1 * value2);
      // Fully fused multiply-add and halve
      result_value = (3.0 + (value1 * value2)) / 2.0;
      if result_value == 0.0 then
        // Sign of exact zero result depends on rounding mode
        sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(sign);
      else
        result = FPRound(result_value, fpcr, rounding, fpexc);

  return result;

```

## Library pseudocode for aarch64/

functions/~~memoryfusedrstep/~~**AArch64.AccessIsTagChecked**~~FPRecipStepFused~~

```
// AArch64.AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.
// FPRecipStepFused()
// =====

boolean bits(N) AArch64.AccessIsTagChecked(bits(64) vaddr, FPRecipStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done; AccType FPCRTYPE acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    if fpcr = FPCR[];
    op1 = EffectiveTBI FPNeg(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;
    (op1);

    if boolean altfp = EffectiveTCMAHaveAltFP(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' ||
        return FALSE;
    ) && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

    if ! (type1,sign1,value1) = AArch64.AllocationTagAccessIsEnabled FPUnpack(acctype) then
        return FALSE;

    if acctype IN {(op1, fpcr, fpexc);
    (type2,sign2,value2) = AccType_IFETCH FPUnpack,{op2, fpcr, fpexc};
    (done,result) = AccType_TTW FPProcessNaNs,(type1, type2, op1, op2, fpcr, FALSE, fpexc); AccType_DCFPR
    return FALSE;
    (fpcr);

    if acctype == if !done then
        inf1 = (type1 == AccType_NV2REGISTER FPType_Infinity then
        return FALSE;

    if PSTATE.TC0 == '1' then
        return FALSE;

    if !);
    inf2 = (type2 ==);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo('0');
    elsif inf1 || inf2 then
        result = FPInfinity(sign1 EOR sign2);
    else
        // Fully fused multiply-add
        result_value = 2.0 + (value1 * value2);
        if result_value == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign);
        else
            result = FPRoundIsTagCheckedInstruction FPType_Infinity() then
            return FALSE;
    (result_value, fpcr, rounding, fpexc);

    return TRUE; return result;
```

## Library pseudocode for aarch64/functions/

memory/**AArch64.AddressWithAllocationTag****AArch64.AccessIsTagChecked**

```
// AArch64.AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.
// AArch64.AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

bits(64)boolean AArch64.AddressWithAllocationTag(bits(64) address, AArch64.AccessIsTagChecked(bits(64) va
bits(64) result = address;
bits(4) tag;
acctype)
if PSTATE.M<4> == '1' then return FALSE;

if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
return FALSE;

if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') th
return FALSE;

if !AArch64.AllocationTagAccessIsEnabled(acctype) then
return FALSE;

if acctype IN {AccType_IFETCH, AccType_TTW, AccType_DC, AccType_IC} then
return FALSE;

if acctype == AccType_NV2REGISTER then
return FALSE;

if PSTATE.TC0 == '1' then
return FALSE;

if !IsTagCheckedInstruction(acctype) then
tag = allocation_tag;
else
tag = '0000';
result<59:56> = tag;
return result;() then
return FALSE;

return TRUE;
```

## Library pseudocode for aarch64/functions/

memory/**AArch64.AllocationTagFromAddress****AArch64.AddressWithAllocationTag**

```
// AArch64.AllocationTagFromAddress()
// AArch64.AddressWithAllocationTag()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(4)bits(64) AArch64.AllocationTagFromAddress(bits(64) tagged address)
return tagged_address<59:56>; AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits
bits(64) result = address;
bits(4) tag;
if AArch64.AllocationTagAccessIsEnabled(acctype) then
tag = allocation_tag;
else
tag = '0000';
result<59:56> = tag;
return result;
```

## Library pseudocode for aarch64/functions/ memory/AArch64.CheckAlignmentAArch64.AllocationTagFromAddress

```
// AArch64.CheckAlignment()
// =====
// AArch64.AllocationTagFromAddress()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

boolean bits(4) AArch64.CheckAlignment(bits(64) address, integer alignment, AArch64.AllocationTagFromAddress
    return tagged_address<59:56>; AccType acctype,
        boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
        AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
        AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
    vector = acctype == AccType_VEC;
    boolean check;
    if SCTLRL.A == '1' then check = TRUE;
    elsif HavelSE2Ext() then
        check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLRL.nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

## Library pseudocode for aarch64/functions/memory/AArch64.CheckTagAArch64.CheckAlignment

```
// AArch64.CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed
// AArch64.CheckAlignment()
// =====

boolean AArch64.CheckTag(AArch64.CheckAlignment(bits(64) address, integer alignment, AddressDescriptorAccType
    boolean iswrite)

    aligned = (address == AccessDescriptorAlign accdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattr.tagged then
        (memstatus, readtag) = (address, alignment);
    atomic = acctype IN { PhysMemTagReadAccType_ATOMIC(memaddrdesc, accdesc);
        if, IsFaultAccType_ATOMICRW(memstatus) then,
            ,
            AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
        AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
    vector = acctype == AccType_VEC;
    boolean check;
    if SCTLRL.A == '1' then check = TRUE;
    elsif HavelSE2Ext() then
        check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLRL.nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AlignmentFaultHandleExternalReadAbortAccType_ORDEREDATOMIC(memstatus, mema
        return ptag == readtag;
    else
        return TRUE;(acctype, iswrite, secondstage));

    return aligned;
```



```

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.
// AArch64.CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

bits(size*8) boolean AArch64.MemSingle(bits(64) address, integer size, AArch64.CheckTag( AccType acctype, b
    boolean ispair = FALSE;
    return AArch64.MemSingle[address, size, acctype, aligned, ispair];

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle(bits(64) address, integer size, AccType acctype, boolean aligned, boolean
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = memaddrdesc, AArch64.TranslateAddressAccessDescriptor(address, acctype, iswrite, aligned
    // Check for aborts or debug exceptions
    if accdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        (memstatus, readtag) = IsFaultPhysMemTagRead(memaddrdesc) then (memaddrdesc, accdesc);
    if
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswri
    PhysMemRetStatus memstatus;
    if atomic then
        (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        bits(halfsize * 8) lowhalf, highhalf;
        (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

        value = highhalf:lowhalf;
    else
        for i = 0 to size-1
            (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return value;

```

```

// AArch64.MemSingle[] - assignment (write) form
// =====

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

    PhysMemRetStatus memstatus;
    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite);
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        bits(halfsize*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;

        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
    return ptag == readtag;
else
    return TRUE;

```



```

// AArch64.MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.
// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(4)bits(size*8) AArch64.MemTag[bits(64) address, AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean iswrite, boolean ispair = FALSE;
return
AArch64.MemSingle[address, size, acctype, aligned, ispair];

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean iswrite, boolean ispair = FALSE;
assert size IN {1, 2, 4, 8, 16};
constant halfsize = size DIV 2;
if HaveLSE2Ext() then
    assert CheckAllInAlignedQuantity(address, size, 16);
else
    assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bits(4) value;
bits(size*8) value;
iswrite = FALSE;

iswrite = FALSE;
aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, (address, acctype, iswrite, aligned));
// Check for aborts or debug exceptions
if
    TAG_GRANULEIsFault;

accdesc = (memaddrdesc) then AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
// Check for aborts or debug exceptions
if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

(atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite, memstatus);
PhysMemRetStatus memstatus;
if atomic then
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memaddrdesc) then(memstatus) then
        AArch64.AbortHandleExternalReadAbort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if(memstatus, memaddrdesc, size, accdesc);
elseif splitpair then
    assert ispair;
    bits(halfsize * 8) lowhalf, highhalf;
    (memstatus, lowhalf) = AArch64.AllocationTagAccessIsEnabledPhysMemRead(acctype) && memaddrdesc.memattrs;
    (memstatus, tag) = (memaddrdesc, halfsize, accdesc);
    if PhysMemTagReadIsFault(memaddrdesc, accdesc);
    if(memstatus) then HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return tag;
(memstatus, memaddrdesc, halfsize, accdesc);

value = highhalf:lowhalf;

```

```

else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory. for i = 0 to size-1
    (memstatus, value<8*i+7:8*i>) =

PhysMemRead(memaddrdesc, 1, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    memaddrdesc.address.address = memaddrdesc.address.address + 1;
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =====

AArch64.MemTag[bits(64) address, AArch64.MemSingle[bits(64) address, integer size, AccType acctype] = bits(
    boolean ispair = FALSE;
    AddressDescriptor AArch64.MemSingle memaddrdesc;
    iswrite = TRUE;
[address, size, acctype, aligned, ispair] = value;
    return;

// Stores of allocation tags must be aligned
if address != // AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes. AArch64.MemSingle[bits(64) address, integer size
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, (address, size); TAG_GRANULEAddressDescriptor) then
            boolean secondstage = FALSE; memaddrdesc;
            iswrite = TRUE;

memaddrdesc =
    AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, (address, memaddrdesc.fault));

// Effect on exclusives
if memaddrdesc.memattrs.shareability != AlignmentFaultShareability_NSH(acctype, iswrite, secondstage)

aligned = TRUE;
memaddrdesc = then AArch64.TranslateAddressClearExclusiveByAddress(address, acctype, iswrite, aligned,
    TAG_GRANULEProcessorID);
(), size);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == // Memory array access
accdesc = MemType_DeviceCreateAccessDescriptor then
    c = (acctype);
    if ConstrainUnpredictableHaveMTE2Ext(()) then
        if Unpredictable_DEVICE_TAGSTORE AArch64.AccessIsTagChecked);
        assert c IN {(Constraint_NONEZeroExtend, (address, 64), acctype) then
            bits(4) ptag = Constraint_FAULTAArch64.PhysicalTag};
        if c == (Constraint_FAULTZeroExtend then
            boolean secondstage = FALSE; (address, 64));
        if !
            AArch64.AbortAArch64.CheckTag(address, (memaddrdesc, accdesc, ptag, iswrite) then AlignmentFault

// Check for aborts or debug exceptions
if (ZeroExtend(address, 64), acctype, iswrite);

PhysMemRetStatus memstatus;

```

```

(atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite);
if atomic then
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memaddrdesc) then (memstatus) then
        AArch64.AbortHandleExternalWriteAbort(address, memaddrdesc.fault);
(memstatus, memaddrdesc, size, accdesc);
elseif splitpair then
    assert ispair;
    bits(halfsize*8) lowhalf, highhalf;
    <highhalf, lowhalf> = value;

    accdesc = memstatus = CreateAccessDescriptorPhysMemWrite(acctype);
    // Memory array access
    if (memaddrdesc, halfsize, accdesc, lowhalf);
        if AArch64.AllocationTagAccessIsEnabledIsFault(acctype) && memaddrdesc.memattrs.tagged then
            memstatus = (memstatus) then (memstatus, memaddrdesc, halfsize, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
            memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        else
            for i = 0 to size-1
                memstatus = PhysMemWritePhysMemTagWriteHandleExternalWriteAbort(memaddrdesc, accdesc, value);
            if (memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
                if IsFault(memstatus) then
                    HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc); (memstatus, memaddrdesc, 1, accdesc);
                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;

```

## Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTagAArch64.MemTag

```
// AArch64.PhysicalTag()
// =====
// Generate a Physical Tag from a Logical Tag in an address
// AArch64.MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
return vaddr<59:56>; AArch64.MemTag[bits(64) address, AccType acctype]
AddressDescriptor memaddrdesc;
bits(4) value;

iswrite = FALSE;
aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
TAG_GRANULE);
accdesc = CreateAccessDescriptor(acctype);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return tag;
else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
AddressDescriptor memaddrdesc;
iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
TAG_GRANULE);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType_Device then
    c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
    assert c IN {Constraint_NONE, Constraint_FAULT};
    if c == Constraint_FAULT then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

accdesc = CreateAccessDescriptor(acctype);
// Memory array access
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
```

## Library pseudocode for aarch64/functions/ memory/**AArch64.TranslateAddressForAtomicAccess****AArch64.PhysicalTag**

```
// AArch64.TranslateAddressForAtomicAccess()
// =====
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.
// AArch64.PhysicalTag()
// =====
// Generate a Physical Tag from a Logical Tag in an address

AddressDescriptor bits(4) AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned =AArch64.PhysicalTag(bits(64) vaddr);
    return vaddr<59:56>; AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite,
                                           aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);

    return memaddrdesc;
```

## Library pseudocode for aarch64/functions/

### memory/AddressSupportsLS64AArch64.TranslateAddressForAtomicAccess

```
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean// AArch64.TranslateAddressForAtomicAccess()
// =====
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AddressSupportsLS64(bits(64) address);AArch64.TranslateAddressForAtomicAccess(bits(64)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned =AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite,
        aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.address, ProcessorID(), size);

    if HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);

    return memaddrdesc;
```

## Library pseudocode for aarch64/functions/

### memory/CheckAllInAlignedQuantityAddressSupportsLS64

```
// CheckAllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    returnAddressSupportsLS64(bits(64) address); Align(address+size-1, alignment) == Align(address, align
```

```

// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state. // CheckAllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.
boolean

CheckSPAlignment()
    bits(64) sp = CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return SPAlign[];
    boolean stack_align_check;
    if PSTATE.EL == (address+size-1, alignment) == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;(address, alignment);

```

Library pseudocode for aarch64/functions/  
memory/CheckSingleAccessAttributesCheckSPAlignment

```
// CheckSingleAccessAttributes()
// =====
//
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED UNPREDICTABLE
// if they generate an alignment fault. If they do not generate an alignment fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
//
// The function returns (atomic, splitpair), where
//   atomic indicates if the access is single copy atomic.
//   splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
//   when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
//   as byte accesses.

(boolean, boolean) // CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state. CheckSingleAccessAttributes(bits(64) address,
bits(64) sp = MemoryAttributesSP memattrs, integer size, []);
boolean stack_align_check;
if PSTATE.EL == EL0
    AccTypeEL0 acctype, boolean iswrite, boolean aligned, boolean ispair)
isnormalwb = (memattrs.memtype == then
stack_align_check = ( MemType_NormalSCTLR &&
    memattrs.inner.attrs == [].SA0 != '0');
else
stack_align_check = ( MemAttr_WBSCTLR &&
    memattrs.outer.attrs == [].SA != '0');

if stack_align_check && sp != MemAttr_WBAlign);

atomic = TRUE;
splitpair = FALSE;
if isnormalwb then return (atomic, splitpair);

accatomic = acctype IN {(sp, 16) then AccType_ATOMICAARCH64.SPAlignmentFault, AccType_ATOMICRW, AccType_ATOMICSCTLR,
AccType_ORDEREDATOMICSCTLR, AccType_ATOMICSCTLR, AccType_A32LSMD};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDRW };

if !aligned && (accatomic || ordered) then
    atomic = ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC);
    if !atomic then
        secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
    else
        return (atomic, splitpair);
();

if ispair && aligned then
    // load / store pair requests that are aligned to each register access are split into 2 single copy atomic
    atomic = FALSE;
    splitpair = TRUE;
    return (atomic, splitpair);

if aligned then
    return (atomic, splitpair);

atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not NO
return (atomic, splitpair); return;
```

## Library pseudocode for aarch64/functions/ memory/IsTagCheckedInstructionCheckSingleAccessAttributes

```
// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.
boolean// CheckSingleAccessAttributes()
// =====
//
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED UNPRED
// if they generate an alignment fault. If they do not generate an alignment fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
//
// The function returns (atomic, splitpair), where
// atomic indicates if the access is single copy atomic.
// splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
// when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
// as byte accesses.

(boolean, boolean) IsTagCheckedInstruction(); CheckSingleAccessAttributes(bits(64) address, MemoryAttributes
    AccType acctype, boolean iswrite, boolean aligned, boolean ispair)
    isnormalwb = (memattrs.memtype == MemType_Normal &&
        memattrs.inner.attrs == MemAttr_WB &&
        memattrs.outer.attrs == MemAttr_WB);

    atomic = TRUE;
    splitpair = FALSE;
    if isnormalwb then return (atomic, splitpair);

    accatomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
        AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC };

    if !aligned && (accatomic || ordered) then
        atomic = ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC);
        if !atomic then
            secondstage = FALSE;
            AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
        else
            return (atomic, splitpair);

    if ispair && aligned then
        // load / store pair requests that are aligned to each register access are split into 2 single copy
        atomic = FALSE;
        splitpair = TRUE;
        return (atomic, splitpair);

    if aligned then
        return (atomic, splitpair);

    atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not No
    return (atomic, splitpair);
```



```

// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) // Returns True if the current instruction uses tag-checked memory access,
// False otherwise.
boolean Mem(bits(64) address, integer size, IsTagCheckedInstruction(), AccType acctype)
    boolean ispair = FALSE;
    return Mem[address, size, acctype, ispair];

bits(size*8) Mem(bits(64) address, integer size, AccType acctype, boolean ispair)
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    bits(halfsize * 8) lowhalf, highhalf;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    boolean atomic;
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif acctype IN {AccType_VEC, AccType_VECSTREAM} then
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_pair = FALSE;
        single_is_aligned = TRUE;
        lowhalf = AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, single_is_pair];
        highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, single_is_pair];
        value = highhalf:lowhalf;
    elseif atomic && ispair then
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
    elseif !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
        elseif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
            lowhalf = AArch64.MemSingle[address, halfsize, acctype, aligned, ispair];
            highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair];
            value = highhalf:lowhalf;
        else
            value = AArch64.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

```

```

return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem[address, size, acctype, ispair] = value_in;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
    boolean atomic;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif (acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_aligned = TRUE;
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, ispair] = highhalf;
    elseif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    elseif !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elseif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
    else
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

```



```

// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.
// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size)bits(size*8) MemAtomic(bits(64) address, Mem[bits(64) address, integer size, MemAtomicOp op, bit
    boolean ispair = FALSE;
    return Mem[address, size, acctype, ispair];

bits(size*8) Mem[bits(64) address, integer size, AccType stacctype)
    bits(size) newvalue;
    memaddrdesc =acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    bits(halfsize * 8) lowhalf, highhalf;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.TranslateAddressForAtomicAccessAArch64.CheckAlignment(address, size);
    ldaccdesc =(address, halfsize, acctype, iswrite);
    else
        aligned = CreateAccessDescriptorAArch64.CheckAlignment(ldacctype);
    staccdesc =(address, size, acctype, iswrite);
    boolean atomic;
    if size != 16 || !(acctype IN { CreateAccessDescriptorAccType_VEC(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) =, PhysMemReadAccType_VECSTREAM(memaddrdesc, size DIV 8, ldaccdesc);
    if}} then
        if ! IsFaultHaveLSE2Ext(memstatus) then() then
            atomic = aligned;
        else
            atomic =
                HandleExternalReadAbortCheckAllInAlignedQuantity(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if(address, size, 16);
    elsif acctype IN { BigEndianAccType_VEC(ldacctype) then
        oldvalue =, BigEndianReverseAccType_VECSTREAM(oldvalue);

    case op of
        when} then
            // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
            // 64-bit aligned.
            atomic = address == MemAtomicOp_ADDAlign    newvalue = oldvalue + value;
            when(address, 8);
        else
            // 16-byte integer access
            atomic = address == MemAtomicOp_BICAlign    newvalue = oldvalue AND NOT(value);
            when(address, 16);

        if !atomic && ispair && address == MemAtomicOp_EORAlign    newvalue = oldvalue EOR value;
            when(address, halfsize) then
                single_is_pair = FALSE;
                single_is_aligned = TRUE;
                lowhalf = MemAtomicOp_ORRAArch64.MemSingle    newvalue = oldvalue OR value;
            when[address, halfsize, acctype, single is aligned, single is pair];
            highhalf = MemAtomicOp_SMAXAArch64.MemSingle    newvalue = if[address + halfsize, halfsize, acctype]
            value = highhalf:lowhalf;
        elsif atomic && ispair then
            value = SIntAArch64.MemSingle(oldvalue) >[address, size, acctype, aligned, ispair];
        elsif !atomic then

            assert size > 1;
            value<7:0> = SIntAArch64.MemSingle(value) then oldvalue else value;
            when[address, 1, acctype, aligned];

```

```

// For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
if !aligned then
    c = MemAtomicOp_SMINConstrainUnpredictable newvalue = if( SIntUnpredictable_DEVPAGE2(oldvalue)
    assert c IN { SIntConstraint_FAULT(value) then value else oldvalue;
    when, MemAtomicOp_UMAXConstraint_NONE newvalue = if};
    if c == UIntConstraint_NONE(oldvalue) >then aligned = TRUE;

    for i = 1 to size-1
        value<8*i+7:8*i> = UIntAArch64.MemSingle(value) then oldvalue else value;
        when[address+i, 1, acctype, aligned];
    elsif size == 16 && acctype IN { MemAtomicOp_UMINAccType_VEC newvalue = if, UIntAccType_VECSTREAM(o
        lowhalf = UIntAArch64.MemSingle(value) then value else oldvalue;
        when[address, halfsize, acctype, aligned, ispair];
        highhalf = MemAtomicOp_SWPAArch64.MemSingle newvalue = value;

    if[address + halfsize, halfsize, acctype, aligned, ispair];
    value = highhalf:lowhalf;
    else
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(stacctype) then
        newvalue =(acctype) then
        value = BigEndianReverse(newvalue);
        memstatus =(value);

    return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem[address, size, acctype, ispair] = value_in;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
    boolean atomic;
    boolean aligned;
    if PhysMemWriteBigEndian(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if(acctype) then
        value = IsFaultBigEndianReverse(memstatus) then(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned =
        (address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        atomic = CheckAllInAlignedQuantity(address, size, 16);
    elsif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elsif (acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

```

```

if !atomic && ispair && address == Align(address, halfsize) then
    single_is_aligned = TRUE;
    <highhalf, lowhalf> = value;
    AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, ispair] = lowhalf;
    AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, ispair] = highhalf;
elseif atomic && ispair then
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
elseif !atomic then
    assert size > 1;
    AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    if !aligned then
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elseif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
    else
        AArch64.MemSingleHandleExternalWriteAbortAArch64.CheckAlignment(memstatus, memaddrdesc, size DIV

// Load operations return the old (pre-operation) value
return oldvalue;[address, size, acctype, aligned, ispair] = value;
return;

```

```

// MemAtomicCompareAndSwap()
// =====
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.
// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                   bits(size) newvalue_in, MemAtomic(bits(64) address, MemAtomicOp_op, bi
    bits(size) newvalue = newvalue_in;
bits(size) newvalue;
memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
ldaccdesc = CreateAccessDescriptor(ldacctype);
staccdesc = CreateAccessDescriptor(stacctype);

// All observers in the shareability domain observe the
// following load and store atomically.
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
if BigEndian(ldacctype) then
    oldvalue = BigEndianReverse(oldvalue);

case op of
    when MemAtomicOp_ADD    newvalue = oldvalue + value;
    when MemAtomicOp_BIC    newvalue = oldvalue AND NOT(value);
    when MemAtomicOp_EOR    newvalue = oldvalue EOR value;
    when MemAtomicOp_ORR    newvalue = oldvalue OR value;
    when MemAtomicOp_SMAX   newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
    when MemAtomicOp_SMIN   newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
    when MemAtomicOp_UMAX   newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
    when MemAtomicOp_UMIN   newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
    when MemAtomicOp_SWP    newvalue = value;

if oldvalue == expectedvalue then
    if if BigEndian(stacctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);

// Load operations return the old (pre-operation) value
return oldvalue;

```



```

// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.
// MemAtomicCompareAndSwap()
// =====
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(512)bits(size) MemLoad64B(bits(64) address, MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
bits(size) newvalue_in, AccType acctype)

    bits(512) data;
    boolean iswrite = FALSE;
    constant integer size = 64;

    aligned = ldacctype, AArch64.CheckAlignmentAccType(address, size, acctype, iswrite);

    if !stacctype)
        bits(size) newvalue = newvalue_in;
    memaddrdesc = AddressSupportsLS64AArch64.TranslateAddressForAtomicAccess(address) then
        c = (address, size);
    ldaccdesc = ConstrainUnpredictableCreateAccessDescriptor({ldacctype});
    staccdesc = Unpredictable_LS64UNSUPPORTEDCreateAccessDescriptor);
    assert c IN {(stacctype)};

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) = Constraint_LIMITED_ATOMICITYPhysMemRead, (memaddrdesc, size DIV 8, ldaccdesc);
    if Constraint_FAULT};

    if c == Constraint_FAULT then
        // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
        boolean secondstage = FALSE;
        boolean s2fslwalk = FALSE;
        FaultRecord fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fslwalk);
        AArch64.Abort(address, fault);
    else
        // Accesses are not single-copy atomic above the byte level
        for i = 0 to 63
            data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
        return data;

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then (memstatus) then
        AArch64.AbortHandleExternalReadAbort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != (memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if Shareability_NSHBigEndian then (ldacctype) then
        oldvalue =
            ClearExclusiveByAddressBigEndianReverse(memaddrdesc.paddress, {oldvalue});

    if oldvalue == expectedvalue then
        if ProcessorIDBigEndian(), size);

    // Memory array access
    accdesc = (stacctype) then
        newvalue = CreateAccessDescriptorBigEndianReverse(acctype);

    if (newvalue);
        memstatus = HaveMTE2ExtPhysMemWrite() then
    (memaddrdesc, size DIV 8, staccdesc, newvalue);
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, acctype, iswrite);

```

```

PhysMemRetStatus memstatus;
(memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
    HandleExternalReadAbortHandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
return data; (memstatus, memaddrdesc, size DIV 8, staccdesc);
return oldvalue;

```

## Library pseudocode for aarch64/functions/memory/MemStore64BMemLoad64B

```
// MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

bits(512)

MemStore64B(bits(64) address, bits(512) value, MemLoad64B(bits(64) address, AccType acctype)
    boolean iswrite = TRUE;
    bits(512) data;
    boolean iswrite = FALSE;
    constant integer size = 64;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fslwalk = FALSE;
            fault = boolean s2fslwalk = FALSE; FaultRecord fault = AArch64.ExclusiveFault(acct
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to 63 // Accesses are not single-copy atomic above the byte level
                for i = 0 to 63
                    data<7+8*i : 8*i> =
                        AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
            else
                -=[address+8*i, 1, acctype, aligned];
                return data; memaddrdesc;
            memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

            // Check for aborts or debug exceptions
            if IsFault(memaddrdesc) then
                AArch64.Abort(address, memaddrdesc.fault);

            // Effect on exclusives
            if memaddrdesc.memattrs.shareability != Shareability_NSH then
                ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

            // Memory array access
            accdesc = CreateAccessDescriptor(acctype);
            if HaveMTE2Ext() then
                if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
                    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
                    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                        AArch64.TagCheckFault(address, acctype, iswrite);

            PhysMemRetStatus memstatus;
            (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbortMemStore64BWithRetAddressDescriptor(address, value, acctype); // Return s
            return;(memstatus, memaddrdesc, size, accdesc);
            return data;
```

```

// MemStore64BWithRet()
// =====
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) // MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store. MemStore64BWithRet(bits(64) address, bits(512) value, MemStore64B(L
boolean iswrite = TRUE;
constant integer size = 64;
aligned =
    AddressDescriptor memaddrdesc;
boolean iswrite = TRUE;
constant integer size = 64;

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
memaddrdesc =
if ! AArch64.TranslateAddressAddressSupportsLS64(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if(address) then
    c = IsFaultConstrainUnpredictable(memaddrdesc) then{
        AArch64.AbortUnpredictable_LS64UNSUPPORTED(address, memaddrdesc.fault);
    return};
    assert c IN { ZeroExtendConstraint_LIMITED_ATOMICITY('1')};

// Effect on exclusives
if memaddrdesc.memattrs.shareability !=, Shareability_NSHConstraint_FAULT then};

if c ==
    ClearExclusiveByAddressConstraint_FAULT(memaddrdesc.paddress, then
        // Generate a Data Abort reported using the DFSC code of 110101.
        boolean secondstage = FALSE;
        boolean s2fslwalk = FALSE;
        fault = ProcessorIDAArch64.ExclusiveFault(), 64);

// Memory array access
accdesc = (acctype, iswrite, secondstage, s2fslwalk); CreateAccessDescriptorAArch64.Abort(acctype);

if(address, fault);
else
    // Accesses are not single-copy atomic above the byte level.
    for i = 0 to 63 HaveMTE2ExtAArch64.MemSingle() then
        if[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
else
    == AArch64.AccessIsTagCheckedMemStore64BWithRet(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, acctype, iswrite);
            return ZeroExtend('1');

memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
return memstatus.store64bstatus; {address, value, acctype}; // Return status is ignored by ST64B
return;

```

## Library pseudocode for aarch64/functions/ memory/MemStore64BWithRetStatusMemStore64BWithRet

```
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.
// MemStore64BWithRet()
// =====
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRetStatus(); MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype,
AddressDescriptor memaddrdesc;
boolean iswrite = TRUE;
constant integer size = 64;

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
    return ZeroExtend('1');

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);

if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, acctype, iswrite);
            return ZeroExtend('1');

memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
return memstatus.store64bstatus;
```

## Library pseudocode for aarch64/functions/memory/**NVMemMemStore64BWithRetStatus**

```
// NVMem[] - non-assignment form
// =====
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.
bits(64) NVMem[integer offset]
    assert offset > 0;
    bits(64) address = MemStore64BWithRetStatus(); SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// =====
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;
```

## Library pseudocode for aarch64/functions/memory/PhysMemTagReadNVMem

```
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
(PhysMemRetStatus, bits(4))// NVMem[] - non-assignment form
// =====
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

bits(64) PhysMemTagRead(NVMem[integer offset])
    assert offset > 0;
    bits(64) address = AddressDescriptorSignExtend desc, (VNCR_EL2.BADDR:offset<11:0>, 64);
    return [address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// =====
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    bits(64) address = SignExtendAccessDescriptorMem accdesc); (VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;
```

**Library pseudocode for aarch64/functions/memory/PhysMemTagWritePhysMemTagRead**

```
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
PhysMemRetStatus(PhysMemRetStatus, bits(4)) PhysMemTagWrite(PhysMemTagRead(AddressDescriptor desc, AccessType accessType, bits(4)) tag, bits(4) data)
```

## Library pseudocode for aarch64/functions/memory/SetTagCheckedInstructionPhysMemTagWrite

```
// Flag the current instruction as using/not using memory tag checking.// This is the hardware operation.
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
PhysMemRetStatus
SetTagCheckedInstruction(boolean checked);PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdes
```

## Library pseudocode for aarch64/functions/mopsmemory/CPYPostSizeChoiceSetTagCheckedInstruction

```
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64)// Flag the current instruction as using/not using memory tag checking. CPYPostSizeChoice(bits(64)
```

## Library pseudocode for aarch64/functions/mops/CPYPreSizeChoiceCPYPostSizeChoice

```
// Returns the size of the copy that is performed by the CPYP* instructions for this
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);CPYPostSizeChoice(bits(64)
```

## Library pseudocode for aarch64/functions/mops/CPYSizeChoiceCPYPreSizeChoice

```
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.
integer// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);CPYPreSizeChoice(bits(64)
```

## Library pseudocode for aarch64/functions/mops/CheckMOPSEnabledCPYSizeChoice

```
// CheckMOPSEnabled()
// =====
// Check for EL0 and EL1 access to the CPY* and SET* instructions.// Returns the size of the block this p
// parameters of the destination, source and size of the copy.
integer

CheckMOPSEnabled()
    if (PSTATE.EL IN {CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);EL0, EL1}
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if (PSTATE.EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
        (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

## Library pseudocode for aarch64/functions/mops/MOPSStageCheckMOPSEnabled

```
enumeration // CheckMOPSEnabled()
// =====
// Check for EL0 and EL1 access to the CPY* and SET* instructions. MOPSStage {CheckMOPSEnabled()
    if (PSTATE.EL IN { MOPSStage_Prologue, MOPSStage_Main, } && () &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if (PSTATE.EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
        (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHostMOPSStage_Epilogue };() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

## Library pseudocode for aarch64/functions/mops/MaxBlockSizeCopiedBytesMOPSStage

```
// MaxBlockSizeCopiedBytes()
// =====
// Returns the maximum number of bytes that can used in a single block of the copy.

integer enumeration MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy"; MOPSStage {MOP
```

## Library pseudocode for aarch64/functions/mops/MemCpyAccessTypesMaxBlockSizeCopiedBytes

```
// MemCpyAccessTypes()
// =====
// Return the read and write access types for a CPY* instruction.
// MaxBlockSizeCopiedBytes()
// =====
// Returns the maximum number of bytes that can used in a single block of the copy.

(AccType, AccType) integer MemCpyAccessTypes(bits(4) options)
    unpriv_at_el1 = PSTATE.EL == MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy"; EL1 && !(EL2
        HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

    runpriv_at_el1 = options<1> == '1' && unpriv_at_el1;
    runpriv_at_el2 = options<1> == '1' && unpriv_at_el2;
    wunpriv_at_el1 = options<0> == '1' && unpriv_at_el1;
    wunpriv_at_el2 = options<0> == '1' && unpriv_at_el2;

    user_access_override = HaveUA0Ext() && PSTATE.UA0 == '1';

    AccType racctype;
    if !user_access_override && (runpriv_at_el1 || runpriv_at_el2) then
        racctype = if options<3> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        racctype = if options<3> == '0' then AccType_NORMAL else AccType_STREAM;

    AccType wacctype;
    if !user_access_override && (wunpriv_at_el1 || wunpriv_at_el2) then
        wacctype = if options<2> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        wacctype = if options<2> == '0' then AccType_NORMAL else AccType_STREAM;

    return (racctype, wacctype);
```

## Library pseudocode for aarch64/functions/mops/MemCpyDirectionChoiceMemCpyAccessTypes

```
// Returns true if in the non-overlapping case of a memcpy of size cpysize bytes
// from the source address fromaddress to destination address toaddress is done
// in the forward direction on this implementation.
boolean MemCpyAccessTypes()
// =====
// Return the read and write access types for a CPY* instruction.

(AccType, AccType) MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize); MemCpyAccessTypes()
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() &&
HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

runpriv_at_el1 = options<1> == '1' && unpriv_at_el1;
runpriv_at_el2 = options<1> == '1' && unpriv_at_el2;
wunpriv_at_el1 = options<0> == '1' && unpriv_at_el1;
wunpriv_at_el2 = options<0> == '1' && unpriv_at_el2;

user_access_override = HaveUA0Ext() && PSTATE.UA0 == '1';

AccType racctype;
if !user_access_override && (runpriv_at_el1 || runpriv_at_el2) then
    racctype = if options<3> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
else
    racctype = if options<3> == '0' then AccType_NORMAL else AccType_STREAM;

AccType wacctype;
if !user_access_override && (wunpriv_at_el1 || wunpriv_at_el2) then
    wacctype = if options<2> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
else
    wacctype = if options<2> == '0' then AccType_NORMAL else AccType_STREAM;

return (racctype, wacctype);
```

## Library pseudocode for aarch64/functions/mops/MemCpyOptionAMemCpyDirectionChoice

```
// MemCpyOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY*/SET* instructions, and FALSE otherwise.

// Returns true if in the non-overlapping case of a memcpy of size cpysize bytes
// from the source address fromaddress to destination address toaddress is done
// in the forward direction on this implementation.
boolean MemCpyOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY*/SET* instructions use Option A"; MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize);
```

## Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedEMemCpyOptionA

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
// MemCpyOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY*/SET* instructions, and FALSE otherwise.

boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress,
bits(64) cpysize); MemCpyOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY*/SET* instructions use Option A";
```

## Library pseudocode for aarch64/functions/

### mops/MemCpyParametersIllformedMMemCpyParametersIllformedE

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddress,
MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress,
bits(64) cpySize);
```

## Library pseudocode for aarch64/functions/

### mops/MemCpyZeroSizeCheckMemCpyParametersIllformedM

```
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyZeroSizeCheck();MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddress,
bits(64) cpySize);
```

## Library pseudocode for aarch64/functions/mops/MemSetAccessTypeMemCpyZeroSizeCheck

```
// MemSetAccessType()
// =====
// Return the access type for a SET* instruction.

AccType// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemSetAccessType(bits(2) options)
    unpriv_at_el1 = options<0> == '1' && PSTATE.EL == MemCpyZeroSizeCheck(); EL1 && !(EL2Enabled() &&
HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = (options<0> == '1' && PSTATE.EL == EL2 &&
HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');

    user_access_override = HaveUA0Ext() && PSTATE.UA0 == '1';

    AccType acctype;
    if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
        acctype = if options<1> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        acctype = if options<1> == '0' then AccType_NORMAL else AccType_STREAM;

    return acctype;
```

## Library pseudocode for aarch64/functions/ mops/MemSetParametersIllformedEMemSetAccessType

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.
boolean MemSetAccessType()
// =====
// Return the access type for a SET* instruction.

AccType MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGE); MemSetAccessType(bits(2) options)
    unpriv_at_el1 = options<0> == '1' && PSTATE.EL == EL1 && !(EL2Enabled() &&
    HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = (options<0> == '1' && PSTATE.EL == EL2 &&
    HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');

    user_access_override = HaveUA0Ext() && PSTATE.UA0 == '1';

    AccType acctype;
    if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
        acctype = if options<1> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        acctype = if options<1> == '0' then AccType_NORMAL else AccType_STREAM;

    return acctype;
```

## Library pseudocode for aarch64/functions/ mops/MemSetParametersIllformedMMemSetParametersIllformedE

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.
boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGM); MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGE);
```

## Library pseudocode for aarch64/functions/ mops/MemSetZeroSizeCheckMemSetParametersIllformedM

```
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.
boolean MemSetZeroSizeCheck(); MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGM);
```

## Library pseudocode for aarch64/functions/ mops/MismatchedCpySetTargetELMemSetZeroSizeCheck

```
// MismatchedCpySetTargetEL()
// =====
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) // Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MismatchedCpySetTargetEL()
    bits(2) target_el;

    if MemSetZeroSizeCheck(); UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```

## Library pseudocode for aarch64/functions/ mops/MismatchedMemCpyExceptionMismatchedCpySetTargetEL

```
// MismatchedMemCpyException()
// =====
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call. // MismatchedCpySetTargetEL()
// =====
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2)

MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
    boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = MismatchedCpySetTargetEL()
    bits(2) target_el;

    if ThisInstrAddrUInt();
    integer vect_offset = 0x0;
    bits(2) target_el = (PSTATE.EL) > MismatchedCpySetTargetELUInt(); {

        ExceptionRecordEL1 exception =) then
            target_el = PSTATE.EL;
        elsif PSTATE.EL == ExceptionSyndromeEL0(&&Exception_MemCpyMemSetEL2Enabled());
            exception.syndrome<24> = '0';
            exception.syndrome<23> = '0';
            exception.syndrome<22:19> = options;
            exception.syndrome<18> = if from_epilogue then '1' else '0';
            exception.syndrome<17> = if wrong_option then '1' else '0';
            exception.syndrome<16> = if option_a then '1' else '0';
            // exception.syndrome<15> is RES0
            exception.syndrome<14:10> = destreg<4:0>;
            exception.syndrome<9:5> = srcreg<4:0>;
            exception.syndrome<4:0> = sizereg<4:0>; () && HCR_EL2.TGE == '1' then
                target_el =

;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1AArch64.TakeExceptionEL2(target_el, exception, preferred_exception_return, vect_offset)

    return target_el;
```

## Library pseudocode for aarch64/functions/ mops/MismatchedMemSetExceptionMismatchedMemCpyException

```
// MismatchedMemSetException()
// MismatchedMemCpyException()
// =====
// Generates an exception for a SET* instruction if the version
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer datareg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(2) options,
                          boolean is_SETG)
MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24> = '1';
    exception.syndrome<23> = if is_SETG then '1' else '0';
    // exception.syndrome<22:21> is RES0
    exception.syndrome<20:19> = options;
    exception.syndrome<24> = '0';
    exception.syndrome<23> = '0';
    exception.syndrome<22:19> = options;
    exception.syndrome<18> = if from_epilogue then '1' else '0';
    exception.syndrome<17> = if wrong_option then '1' else '0';
    exception.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5> = datareg<4:0>;
    exception.syndrome<9:5> = srcreg<4:0>;
    exception.syndrome<4:0> = sizereg<4:0>;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/mops/SETPostSizeChoiceMismatchedMemSetException

```
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64)// MismatchedMemSetException()
// =====
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call. SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, bo
    boolean wrong_option, boolean from_epilogue, bits(2) options,
    boolean is_SETG)
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24> = '1';
    exception.syndrome<23> = if is_SETG then '1' else '0';
    // exception.syndrome<22:21> is RES0
    exception.syndrome<20:19> = options;
    exception.syndrome<18> = if from_epilogue then '1' else '0';
    exception.syndrome<17> = if wrong_option then '1' else '0';
    exception.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5> = datareg<4:0>;
    exception.syndrome<4:0> = sizereg<4:0>;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/mops/~~SETPreSizeChoice~~~~SETPostSizeChoice~~

```
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);
```

## Library pseudocode for aarch64/functions/mops/~~SETSizeChoice~~~~SETPreSizeChoice~~

```
// Returns the size of the block thisperformed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of AlignSize.
integer// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);
```



```

// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) // Returns the size of the block this performed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of AlignSize.
integer AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize); EffectiveTBI(ptr)
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else selbit = if tbi then ptr<55> else ptr<63>;

    if HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
            PAC = 0x0000000000000000<63:0>;

```

```

    elsif !HaveEnhancedPAC2() then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

    // preserve the determination between upper and lower address at bit<55> and insert PAC
    if !HaveEnhancedPAC2() then
        if tbi then
            result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            if tbi then
                result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
            else
                result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR
                    PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
            return result;

```



```

// AddPACDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.
// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
            bits(64) PAC;
            bits(64) result;
            bits(64) ext_ptr;
            bits(64) extfield;
            bit selbit;
            boolean tbi = EL0EffectiveTBI
                boolean IsEL1Regime = (ptr, !data, PSTATE.EL) == '1';
                integer top_bit = if tbi then 55 else 63;

                // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
                // the pointer to select between upper and lower ranges, and preserve this.
                // This handles the awkward case where there is apparently no correct choice between
                // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
                // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
                if PtrHasUpperAndLowerAddrRanges() then
                    assert S1TranslationRegime() == () IN { EL1;
                        Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
                        TrapEL2 = (, EL2EnabledEL2() && HCR_EL2.API == '0' &&
                            (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
                        TrapEL3 =};
                    if HaveELS1TranslationRegime(()) == EL3 && SCR_EL3.API == '0';
                    when EL1
                        Enable = SCTLR_EL1.EnDA;
                        TrapEL2 = then
                            // EL1 translation regime registers
                            if data then
                                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                                    selbit = ptr<55>;
                                else
                                    selbit = ptr<63>;
                            else
                                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                                    selbit = ptr<55>;
                                else
                                    selbit = ptr<63>;
                            else
                                // EL2 translation regime registers
                                if data then
                                    if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                                        selbit = ptr<55>;
                                    else
                                        selbit = ptr<63>;
                                else
                                    if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                                        (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                                        selbit = ptr<55>;
                                    else
                                        selbit = ptr<63>;
                                else selbit = if tbi then ptr<55> else ptr<63>;

```

```

if EL2EnabledHaveEnhancedPAC2() && HCR_EL2.API == '0';
    TrapEL3 =() && HaveELConstPACField() then selbit = ptr<55>;
integer bottom_PAC_bit =EL3CalculateBottomPACBit() && SCR_EL3.API == '0';
    when(selbit);

// The pointer authentication code field takes all the available bits in between
extfield = EL2Replicate
    Enable = SCTLR_EL2.EnDA;
    TrapEL2 = FALSE;
    TrapEL3 =(selbit, 64);

// Compute the pointer authentication code for a ptr with good extension bits
if tbi then
    ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
else
    ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

PAC = HaveELComputePAC((ext_ptr, modifier, K<127:64>, K<63:0>);

// Check if the ptr has good extension bits and corrupt the pointer authentication code if not
if !EL3IsZero) && SCR_EL3.API == '0';
    when(ptr<top_bit:bottom_PAC_bit>) && ! EL3IsOnes
        Enable = SCTLR_EL3.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

if Enable == '0' then
    return X;
elseif TrapEL3 &&(ptr<top_bit:bottom_PAC_bit>) then
    if EL3SDDTrapPriorityHaveEnhancedPAC() then
        UNDEFINED;
    elseif TrapEL2 then
        PAC = 0x0000000000000000<63:0>;
    elseif !
        TrapPACUseHaveEnhancedPAC2() then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// preserve the determination between upper and lower address at bit<55> and insert PAC
if !EL2HaveEnhancedPAC2);
elseif TrapEL3 then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        TrapPACUse(EL3);
else
    return AddPAC(X, Y, APDAKey_EL1, TRUE);() then
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
else
    if tbi then
        result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>):ptr<bottom_PAC
    else
        result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR
            PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
return result;

```

```

// AddPACDB()
// AddPACDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.
// APDAKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
AddPACDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;
    bits(128) APDAKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDB else SCTLRL_EL2.EnDB;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDA else SCTLRL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDB;
            Enable = SCTLRL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDB;
            Enable = SCTLRL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDB;
            Enable = SCTLRL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && if Enable == '0' then return X;
    elsif TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else);
    elsif TrapEL3 then
        TrapPACUse(EL3);
    else
        return AddPAC(X, Y, APDBKey_EL1, TRUE);(X, Y, APDAKey_EL1, TRUE);

```

```

// AddPACGA()
// AddPACDB()
// =====
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
AddPACDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = Enable = SCTL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            Enable = SCTL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            Enable = SCTL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL3 && if Enable == '0' then return X;
    elsif TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if elsif TrapEL3 then Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePACAddPAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>><63:32>; Zeros(32); (X, Y, APDBKey_EL1<127:64>, APDBKey_EL1<63:0>><63:32>; Zeros(32));

```

```

// AddPACIA()
// AddPACGA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
AddPACGA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;
    bits(128) APGAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = TrapEL2 == ( S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
        TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
        TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && if TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if elsif TrapEL3 then Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return (X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:ZerosAddPACComputePAC(X, Y, APIAKey_

```

```

// AddPACIB()
// AddPACIA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
AddPACIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;
    bits(128) APIAKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnIB else SCTL_EL2.EnIB;
            Enable = if IsEL1Regime then SCTL_EL1.EnIA else SCTL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnIB;
            Enable = SCTL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnIB;
            Enable = SCTL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnIB;
            Enable = SCTL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && if Enable == '0' then return X;
    elsif TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else);
    elsif TrapEL3 then
        TrapPACUse(EL3);
    else
        return AddPAC(X, Y, APIBKey_EL1, FALSE); (X, Y, APIAKey_EL1, FALSE);

```

```

// AArch64.PACFailException()
// =====
// Generates a PAC Fail Exception// AddPACIB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64)

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == AddPACIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0 && boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnIB else SCTL_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.TGE == '1';
            bits(64) preferred_exception_return = ( && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = ThisInstrAddrHaveEL();
            vect_offset = 0x0;

            exception = ( ExceptionSyndromeEL3() && SCR_EL3.API == '0';
                when Exception_PACFailEL1);
            exception.syndrome<1:0> = syndrome;
            exception.syndrome<24:2> = Enable = SCTL_EL1.EnIB;
            TrapEL2 = ZerosEL2Enabled(); // RES0

            if() && HCR_EL2.API == '0';
                TrapEL3 = UIntHaveEL(PSTATE.EL) > ( UIntEL3() && SCR_EL3.API == '0';
                when EL0EL2) then Enable = SCTL_EL2.EnIB;
                TrapEL2 = FALSE;
                TrapEL3 =
                    AArch64.TakeExceptionHaveEL(PSTATE.EL, exception, preferred_exception_return, vect_offset);
            elsif route_to_el2 then(
                AArch64.TakeExceptionEL3() && SCR_EL3.API == '0';
                when EL3
                    Enable = SCTL_EL3.EnIB;
                    TrapEL2 = FALSE;
                    TrapEL3 = FALSE;

            if Enable == '0' then return X;
            elsif TrapEL2 then TrapPACUse(EL2, exception, preferred_exception_return, vect_offset);
            else);
            elsif TrapEL3 then
                AArch64.TakeExceptionTrapPACUse();
            else return AddPACEL1EL3, exception, preferred_exception_return, vect_offset); (X, Y, APIBKey_EL1, FA

```



```

// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) // AArch64.PACFailException()
// =====
// Generates a PAC Fail Exception Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit ke
        boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EffectiveTBIEL0(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = && CalculateBottomPACBitEL2Enabled(ptr<55>);
    extfield = () && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ReplicateThisInstrAddr(ptr<55>, 64);
();
    vect_offset = 0x0;

    if tbi then
        original_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = exception = ComputePACExceptionSyndrome(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !(HaveEnhancedPAC2Exception_PACFail()) then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
        if);
    exception.syndrome<1:0> = syndrome;
    exception.syndrome<24:2> = HaveFPACCombinedZeros() || (); // RES0

    if HaveFPAUInt() && !is_combined) then
        if result<54:bottom_PAC_bit> != (PSTATE.EL) > ReplicateUInt(result<55>, (55-bottom_PAC_bit)
            error_code = (if data then '1' else '0'):key_number;
            AArch64.PACFailExceptionEL0(error_code);
    else
        if !() then HaveEnhancedPAC2AArch64.TakeException() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then HaveFPACCombinedAArch64.TakeException() || (HaveFPACEL2() && !is_combined) t
        if result<63:bottom_PAC_bit> !=, exception, preferred_exception_return, vect_offset);
    else ReplicateAArch64.TakeException(result<55>, (64-bottom_PAC_bit)) then
        error_code = (if data then '1' else '0'):key_number;
        AArch64.PACFailExceptionEL1(error_code);
    return result; exception, preferred_exception_return, vect_offset);

```



```

// AuthDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().
// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;
Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
    boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when // Reconstruct the extension field used of adding the PAC to the pointer
            boolean tbi = EL0EffectiveTBI
                boolean IsEL1Regime = (ptr, !data, PSTATE.EL) == '1';
                integer bottom_PAC_bit = S1TranslationRegimeCalculateBottomPACBit() == (ptr<55>);
                extfield = EL1Replicate;
                Enable = if IsEL1Regime then SCTL_EL1.EnDA else SCTL_EL2.EnDA;
                TrapEL2 = ({ptr<55>, 64});

            if tbi then
                original_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
            else
                original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

            PAC = EL2EnabledComputePAC() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0');
            TrapEL3 = (original_ptr, modifier, K<127:64>, K<63:0>);
            // Check pointer authentication code
            if tbi then
                if ! HaveELHaveEnhancedPAC2() then
                    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                        result = original_ptr;
                    else
                        error_code = key_number:NOT(key_number);
                        result = original_ptr<63:55>:error_code:original_ptr<52:0>;
                else
                    result = ptr;
                    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
                    if EL3HaveFPACCombined() && SCR_EL3.API == '0';
                when() || ( EL1HaveFPAC
                    Enable = SCTL_EL1.EnDA;
                    TrapEL2 = () && !is_combined) then
                        if result<54:bottom_PAC_bit> != EL2EnabledReplicate() && HCR_EL2.API == '0';
                    TrapEL3 = (result<55>, (55-bottom_PAC_bit)) then
                        error_code = (if data then '1' else '0'):key_number; HaveELAArch64.PACFailException()
                else
                    if ! EL3HaveEnhancedPAC2() && SCR_EL3.API == '0';
                    when() then
                        if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                            result = original_ptr;
                        else
                            error_code = key_number:NOT(key_number);

```

```

        result = original_ptr<63>:error_code:original_ptr<60:0>;
    else
        result = ptr;
        result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
        result<63:56> = result<63:56> EOR PAC<63:56>;
        if EL2HaveFPACCombined
            Enable = SCTLRL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = () || ( HaveELHaveFPAC(()) && !is_combined) then
                if result<63:bottom_PAC_bit> != EL3Replicate() && SCR_EL3.API == '0';
            when(result<55>, (64-bottom_PAC_bit)) then
                error code = (if data then '1' else '0'):key_number; EL3AArch64.PACFailException
            Enable = SCTLRL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);(error_code);
    return result;

```

## Library pseudocode for aarch64/functions/pac/authdbauthda/AuthDBAuthDA

```
// AuthDB()
// AuthDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;
    bits(128) APDAKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            Enable = if IsEL1Regime then SCTL_EL1.EnDA else SCTL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnDB;
            Enable = SCTL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnDB;
            Enable = SCTL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnDB;
            Enable = SCTL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && if Enable == '0' then return X;
    elsif TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else);
    elsif TrapEL3 then
        TrapPACUse(EL3);
    else
        return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);(X, Y, APDAKey_EL1, TRUE, '0', is_combined)
```

## Library pseudocode for aarch64/functions/pac/authiaauthdb/AuthIAAuthDB

```
// AuthIA()
// AuthDB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthIA(bits(64) X, bits(64) Y, boolean is_combined)
AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;
    bits(128) APDBKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnIA else SCTL_EL2.EnIA;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnIA;
            Enable = SCTL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnIA;
            Enable = SCTL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnIA;
            Enable = SCTL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && if Enable == '0' then return X;
    elsif TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else);
    elsif TrapEL3 then
        TrapPACUse(EL3);
    else
        return Auth(X, Y, APIAKey_EL1, FALSE, '0', is_combined);(X, Y, APDBKey_EL1, TRUE, '1', is_combined)
```

## Library pseudocode for aarch64/functions/pac/authibauthia/AuthIBAuthIA

```
// AuthIB()
// AuthIA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().
// algorithm and key as AddPACIA().

bits(64) AuthIB(bits(64) X, bits(64) Y, boolean is_combined)
AuthIA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;
    bits(128) APIAKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIB else SCTLRL_EL2.EnIB;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIA else SCTLRL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
            when EL1
                Enable = SCTLRL_EL1.EnIB;
                Enable = SCTLRL_EL1.EnIA;
                TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
                TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
            when EL2
                Enable = SCTLRL_EL2.EnIB;
                Enable = SCTLRL_EL2.EnIA;
                TrapEL2 = FALSE;
                TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
            when EL3
                Enable = SCTLRL_EL3.EnIB;
                Enable = SCTLRL_EL3.EnIA;
                TrapEL2 = FALSE;
                TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && if Enable == '0' then return X;
    elsif TrapEL2 then EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else);
    elsif TrapEL3 then
        TrapPACUse(EL3);
    else
        return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);(X, Y, APIAKey_EL1, FALSE, '0', is_combi
```

## Library pseudocode for aarch64/functions/

### pac/calcbottompacbitauthib/CalculateBottomPACBitAuthIB

```
// CalculateBottomPACBit()
// =====
// AuthIB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

integer bits(64) CalculateBottomPACBit(bit top_bit)
integer tsz_field;
boolean using64k; AuthIB(bits(64) X, bits(64) Y, boolean is_combined);
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIBKey_EL1;

APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
case PSTATE.EL of
when
    ConstraintEL0 c;

if boolean IsEL1Regime = PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {() == EL1,};
    Enable = if IsEL1Regime then SCTL_EL1.EnIB else SCTL_EL2.EnIB;
    TrapEL2 = ( EL2EL2Enabled );
    if () && HCR_EL2.API == '0' &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0');
    TrapEL3 = S1TranslationRegimeHaveEL() == ( EL3 ) && SCR_EL3.API == '0';
    when EL1 then
        // EL1 translation regime registers
        tsz_field = if top_bit == '1' then Enable = SCTL_EL1.EnIB;
        TrapEL2 = UIntEL2Enabled(TCR_EL1.T1SZ) else () && HCR_EL2.API == '0';
        TrapEL3 = UInt(TCR_EL1.T0SZ);
        using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
    else
        // EL2 translation regime registers
        assert HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2;
        tsz_field = if top_bit == '1' then Enable = SCTL_EL2.EnIB;
        TrapEL2 = FALSE;
        TrapEL3 = UIntHaveEL(TCR_EL2.T1SZ) else ( UIntEL3(TCR_EL2.T0SZ);
        using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    else
        tsz_field = if PSTATE.EL == ) && SCR_EL3.API == '0';
        when EL2EL3 then Enable = SCTL_EL3.EnIB;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

if Enable == '0' then return X;
elseif TrapEL2 then UIntTrapPACUse(TCR_EL2.T0SZ) else ( UInt(TCR_EL3.T0SZ);
    using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

max_limit_tsz_field = (if !);
elseif TrapEL3 then HaveSmallTranslationTableExtTrapPACUse() then 39 else if using64k then 47 else 48);
if tsz_field > max_limit_tsz_field then
    // TCR_ELx.TySZ is out of range
    c = ( ConstrainUnpredictableEL3 );
else return Unpredictable_RESTnSZAuth;
assert c IN {Constraint_FORCE, Constraint_NONE};
if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
tszmin = if using64k && AArch64.VAMax() == 52 then 12 else 16;
if tsz_field < tszmin then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_NONE};
    if c == Constraint_FORCE then tsz_field = tszmin;
return (64-tsz_field);(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
```

Library pseudocode for aarch64/functions/  
pac/**computepac****calcbottompacbit**/ComputePACCalculateBottomPACBit

```

// ComputePAC()
// =====
// CalculateBottomPACBit()
// =====

bits(64) integer ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64) workingval;
    bits(64) runningmod;
    bits(64) roundkey;
    bits(64) modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    integer iterations;
    if CalculateBottomPACBit(bit_top_bit)
        integer tsz_field;
        boolean using64k; HavePACQARMA3Constraint() then
            iterations = 2;
            RC[0] = 0x0000000000000000<63:0>;
            RC[1] = 0x13198A2E03707344<63:0>;
            RC[2] = 0xA4093822299F31D0<63:0>;
        else
            iterations = 4;
            RC[0] = 0x0000000000000000<63:0>;
            RC[1] = 0x13198A2E03707344<63:0>;
            RC[2] = 0xA4093822299F31D0<63:0>;
            RC[3] = 0x082EFA98EC4E6C89<63:0>;
            RC[4] = 0x452821E638D01377<63:0>;

C:

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to iterations
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = if PACCellShufflePtrHasUpperAndLowerAddRanges(workingval);
            workingval =() then
                assert PACMultS1TranslationRegime(workingval);
                if() IN { HavePACQARMA3EL1() then
                    workingval = PACSub1EL2(workingval);
                else
                    workingval =};
                if PACSubS1TranslationRegime(workingval);
                runningmod =() == TweakShuffleEL1(runningmod<63:0>);
                roundkey = modk0 EOR runningmod;
                workingval = workingval EOR roundkey;
                workingval =then
                    // EL1 translation regime registers
                    tsz_field = if top_bit == '1' then PACCellShuffleUInt(workingval);
                workingval =(TCR_EL1.T1SZ) else PACMultUInt(workingval);
                if(TCR_EL1.T0SZ);
                    using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
                else
                    // EL2 translation regime registers
                    assert HavePACQARMA3HaveEL() then
                        workingval =( PACSub1EL2(workingval);
                else
                    workingval =);
                    tsz_field = if top_bit == '1' then PACSubUInt(workingval);
                workingval =(TCR_EL2.T1SZ) else PACCellShuffleUInt(workingval);
                workingval =(TCR_EL2.T0SZ);
                    using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
                else
                    tsz_field = if PSTATE.EL == PACMultEL2(workingval);
                workingval = key1 EOR workingval;
                workingval =then PACCellInvShuffleUInt(workingval);
                if(TCR_EL2.T0SZ) else HavePACQARMA3UInt() then
                    workingval =(TCR_EL3.T0SZ);

```

```

        using64k = if PSTATE.EL == PACSub1EL2(workingval);
    else
        workingval =then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

    max_limit_tsz_field = (if ! PACInvSubHaveSmallTranslationTableExt(workingval);
    workingval =() then 39 else if using64k then 47 else 48);
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
        c = PACMultConstrainUnpredictable(workingval);
    workingval =({ PACCellInvShuffleUnpredictable_RESTnSZ(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to iterations
        if);
    assert c IN { HavePACQARMA3Constraint_FORCE() then
        workingval =, PACSub1Constraint_NONE(workingval);
    else
        workingval =};
    if c == PACInvSubConstraint_FORCE(workingval);
    if i < iterations then
        workingval =then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && PACMultAArch64.VAMax(workingval);
        workingval =() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = PACCellInvShuffleConstrainUnpredictable(workingval);
        runningmod =({ });
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE{TwakeInvShuffleUnpredictable_RESTnSZ(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;then tsz_field = tszmin;
    return (64-tsz_field);

```



```

// PACCellInvShuffle()
// =====
// ComputePAC()
// =====

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata; ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64) workingval;
    bits(64) runningmod;
    bits(64) roundkey;
    bits(64) modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    integer iterations;
    if HavePACQARMA3() then
        iterations = 2;
        RC[0] = 0x0000000000000000<63:0>;
        RC[1] = 0x13198A2E03707344<63:0>;
        RC[2] = 0xA4093822299F31D0<63:0>;
    else
        iterations = 4;
        RC[0] = 0x0000000000000000<63:0>;
        RC[1] = 0x13198A2E03707344<63:0>;
        RC[2] = 0xA4093822299F31D0<63:0>;
        RC[3] = 0x082EFA98EC4E6C89<63:0>;
        RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to iterations
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        if HavePACQARMA3() then
            workingval = PACSub1(workingval);
        else
            workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if HavePACQARMA3() then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;

```

```

workingval = PACCellInvShuffle(workingval);
if HavePACQARMA3() then
    workingval = PACSub1(workingval);
else
    workingval = PACInvSub(workingval);
workingval = PACMult(workingval);
workingval = PACCellInvShuffle(workingval);
workingval = workingval EOR key0;
workingval = workingval EOR runningmod;
for i = 0 to iterations
    if HavePACQARMA3() then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    if i < iterations then
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

return workingval;

```

### Library pseudocode for aarch64/functions/pac/computepac/PACCellShufflePACCellInvShuffle

```

// PACCellShuffle()
// =====
// PACCellInvShuffle()
// =====

bits(64) PACCellShuffle(bits(64) indata)
PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;

```

```

// PACInvSub()
// =====
// PACCellShuffle()
// =====

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' Toutput<4*i+3:4*i> = '0101';
            when '0001' Toutput<4*i+3:4*i> = '1110';
            when '0010' Toutput<4*i+3:4*i> = '1101';
            when '0011' Toutput<4*i+3:4*i> = '1000';
            when '0100' Toutput<4*i+3:4*i> = '1010';
            when '0101' Toutput<4*i+3:4*i> = '1011';
            when '0110' Toutput<4*i+3:4*i> = '0001';
            when '0111' Toutput<4*i+3:4*i> = '1001';
            when '1000' Toutput<4*i+3:4*i> = '0010';
            when '1001' Toutput<4*i+3:4*i> = '0110';
            when '1010' Toutput<4*i+3:4*i> = '1111';
            when '1011' Toutput<4*i+3:4*i> = '0000';
            when '1100' Toutput<4*i+3:4*i> = '0100';
            when '1101' Toutput<4*i+3:4*i> = '1100';
            when '1110' Toutput<4*i+3:4*i> = '0111';
            when '1111' Toutput<4*i+3:4*i> = '0011';
    return Toutput; PACCellShuffle(bits(64) indata);
bits(64) outdata;
outdata<3:0> = indata<55:52>;
outdata<7:4> = indata<27:24>;
outdata<11:8> = indata<47:44>;
outdata<15:12> = indata<3:0>;
outdata<19:16> = indata<31:28>;
outdata<23:20> = indata<51:48>;
outdata<27:24> = indata<7:4>;
outdata<31:28> = indata<43:40>;
outdata<35:32> = indata<35:32>;
outdata<39:36> = indata<15:12>;
outdata<43:40> = indata<59:56>;
outdata<47:44> = indata<23:20>;
outdata<51:48> = indata<11:8>;
outdata<55:52> = indata<39:36>;
outdata<59:56> = indata<19:16>;
outdata<63:60> = indata<63:60>;
return outdata;

```

```

// PACMult()
// =====
// PACInvSub()
// =====

bits(64) PACMult(bits(64) Sinput)
    bits(4) t0;
    bits(4) t1;
    bits(4) t2;
    bits(4) t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = PACInvSub(bits(64) Tinput)
        // This is a 4-bit substitution from the PRINCE-family cipher
        bits(64) Toutput;
        for i = 0 to 15
            case Tinput<4*i+3:4*i> of
                when '0000' Toutput<4*i+3:4*i> = '0101';
                when '0001' Toutput<4*i+3:4*i> = '1110';
                when '0010' Toutput<4*i+3:4*i> = '1101';
                when '0011' Toutput<4*i+3:4*i> = '1000';
                when '0100' Toutput<4*i+3:4*i> = '1010';
                when '0101' Toutput<4*i+3:4*i> = '1011';
                when '0110' Toutput<4*i+3:4*i> = '0001';
                when '0111' Toutput<4*i+3:4*i> = '1001';
                when '1000' Toutput<4*i+3:4*i> = '0010';
                when '1001' Toutput<4*i+3:4*i> = '0110';
                when '1010' Toutput<4*i+3:4*i> = '1111';
                when '1011' Toutput<4*i+3:4*i> = '0000';
                when '1100' Toutput<4*i+3:4*i> = '0100';
                when '1101' Toutput<4*i+3:4*i> = '1100';
                when '1110' Toutput<4*i+3:4*i> = '0111';
                when '1111' Toutput<4*i+3:4*i> = '0011';
            return Toutput; RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
            t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
            t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
            t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
            t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
            t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
            t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
            t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
            Soutput<4*i+3:4*i> = t3<3:0>;
            Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
            Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
            Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
        return Soutput;

```

```

// PACSub()
// =====
// PACMult()
// =====

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' Toutput<4*i+3:4*i> = '1011';
            when '0001' Toutput<4*i+3:4*i> = '0110';
            when '0010' Toutput<4*i+3:4*i> = '1000';
            when '0011' Toutput<4*i+3:4*i> = '1111';
            when '0100' Toutput<4*i+3:4*i> = '1100';
            when '0101' Toutput<4*i+3:4*i> = '0000';
            when '0110' Toutput<4*i+3:4*i> = '1001';
            when '0111' Toutput<4*i+3:4*i> = '1110';
            when '1000' Toutput<4*i+3:4*i> = '0011';
            when '1001' Toutput<4*i+3:4*i> = '0111';
            when '1010' Toutput<4*i+3:4*i> = '0100';
            when '1011' Toutput<4*i+3:4*i> = '0101';
            when '1100' Toutput<4*i+3:4*i> = '1101';
            when '1101' Toutput<4*i+3:4*i> = '0010';
            when '1110' Toutput<4*i+3:4*i> = '0001';
            when '1111' Toutput<4*i+3:4*i> = '1010';
        return Toutput; PACMult(bits(64) Sinput)
    bits(4) t0;
    bits(4) t1;
    bits(4) t2;
    bits(4) t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;

```

```

// PacSub1()
// =====
// PACSub()
// =====

bits(64) PACSub1(bits(64) Tinput)
    // This is a 4-bit substitution from Qarma signal
PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' Toutput<4*i+3:4*i> = '1010';
            when '0001' Toutput<4*i+3:4*i> = '1101';
            when '0010' Toutput<4*i+3:4*i> = '1110';
            when '0011' Toutput<4*i+3:4*i> = '0110';
            when '0100' Toutput<4*i+3:4*i> = '1111';
            when '0101' Toutput<4*i+3:4*i> = '0111';
            when '0110' Toutput<4*i+3:4*i> = '0011';
            when '0111' Toutput<4*i+3:4*i> = '0101';
            when '1000' Toutput<4*i+3:4*i> = '1001';
            when '1001' Toutput<4*i+3:4*i> = '1000';
            when '1010' Toutput<4*i+3:4*i> = '0000';
            when '1011' Toutput<4*i+3:4*i> = '1100';
            when '1100' Toutput<4*i+3:4*i> = '1011';
            when '1101' Toutput<4*i+3:4*i> = '0001';
            when '1110' Toutput<4*i+3:4*i> = '0010';
            when '1111' Toutput<4*i+3:4*i> = '0100';
            when '0000' Toutput<4*i+3:4*i> = '1011';
            when '0001' Toutput<4*i+3:4*i> = '0110';
            when '0010' Toutput<4*i+3:4*i> = '1000';
            when '0011' Toutput<4*i+3:4*i> = '1111';
            when '0100' Toutput<4*i+3:4*i> = '1100';
            when '0101' Toutput<4*i+3:4*i> = '0000';
            when '0110' Toutput<4*i+3:4*i> = '1001';
            when '0111' Toutput<4*i+3:4*i> = '1110';
            when '1000' Toutput<4*i+3:4*i> = '0011';
            when '1001' Toutput<4*i+3:4*i> = '0111';
            when '1010' Toutput<4*i+3:4*i> = '0100';
            when '1011' Toutput<4*i+3:4*i> = '0101';
            when '1100' Toutput<4*i+3:4*i> = '1101';
            when '1101' Toutput<4*i+3:4*i> = '0010';
            when '1110' Toutput<4*i+3:4*i> = '0001';
            when '1111' Toutput<4*i+3:4*i> = '1010';
    return Toutput;

```

## Library pseudocode for aarch64/functions/pac/computepac/RCPacSub1

```
array bits(64) RC[0..4]; // PacSub1()
// =====

bits(64) PACSub1(bits(64) Tinput)
// This is a 4-bit substitution from Qarma-sigma1
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1010';
        when '0001' Toutput<4*i+3:4*i> = '1101';
        when '0010' Toutput<4*i+3:4*i> = '1110';
        when '0011' Toutput<4*i+3:4*i> = '0110';
        when '0100' Toutput<4*i+3:4*i> = '1111';
        when '0101' Toutput<4*i+3:4*i> = '0111';
        when '0110' Toutput<4*i+3:4*i> = '0011';
        when '0111' Toutput<4*i+3:4*i> = '0101';
        when '1000' Toutput<4*i+3:4*i> = '1001';
        when '1001' Toutput<4*i+3:4*i> = '1000';
        when '1010' Toutput<4*i+3:4*i> = '0000';
        when '1011' Toutput<4*i+3:4*i> = '1100';
        when '1100' Toutput<4*i+3:4*i> = '1011';
        when '1101' Toutput<4*i+3:4*i> = '0001';
        when '1110' Toutput<4*i+3:4*i> = '0010';
        when '1111' Toutput<4*i+3:4*i> = '0100';
    return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/RotCellRC

```
// RotCell()
// =====

bits(4) array bits(64) RC[0..4]; RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRotRotCell

```
// TweakCellInvRot()
// =====
// RotCell()
// =====

bits(4) TweakCellInvRot(bits(4) incell)
RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellRotTweakCellInvRot

```
// TweakCellRot()
// =====
// TweakCellInvRot()
// =====

bits(4) TweakCellRot(bits(4) incell)
TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffleTweakCellRot

```
// TweakInvShuffle()
// =====
// TweakCellRot()
// =====

bits(64)bits(4) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> =TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell; TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

```

// TweakShuffle()
// =====
// TweakInvShuffle()
// =====

bits(64) TweakShuffle(bits(64) indata)
TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = outdata<3:0> = TweakCellRotTweakCellInvRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = (indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellRotTweakCellInvRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = (indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellRotTweakCellInvRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = (indata<31:28>);
    outdata<39:36> = TweakCellRotTweakCellInvRot(indata<63:60>);
    outdata<51:48> = outdata<43:40> = TweakCellRotTweakCellInvRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = (indata<59:56>);
    outdata<47:44> = TweakCellRotTweakCellInvRot(indata<43:40>);
    (indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellRotTweakCellInvRot(indata<39:36>);
    (indata<47:44>);
    return outdata;

```

### Library pseudocode for aarch64/functions/pac/pac/computePac/ConstPACFieldTweakShuffle

```
// ConstPACField()
// =====
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.
// TweakShuffle()
// =====

boolean bits(64) ConstPACField()
    return boolean IMPLEMENTATION_DEFINED "Bit 55 determines the size of the PAC field";TweakShuffle(bits)

bits(64) outdata;
outdata<3:0> = indata<19:16>;
outdata<7:4> = indata<23:20>;
outdata<11:8> = TweakCellRot(indata<27:24>);
outdata<15:12> = indata<31:28>;
outdata<19:16> = TweakCellRot(indata<47:44>);
outdata<23:20> = indata<11:8>;
outdata<27:24> = indata<15:12>;
outdata<31:28> = TweakCellRot(indata<35:32>);
outdata<35:32> = indata<51:48>;
outdata<39:36> = indata<55:52>;
outdata<43:40> = indata<59:56>;
outdata<47:44> = TweakCellRot(indata<63:60>);
outdata<51:48> = TweakCellRot(indata<3:0>);
outdata<55:52> = indata<7:4>;
outdata<59:56> = TweakCellRot(indata<43:40>);
outdata<63:60> = TweakCellRot(indata<39:36>);
return outdata;
```

### Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPACConstPACField

```
// HaveEnhancedPAC()
// =====
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.
// ConstPACField()
// =====
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.

boolean HaveEnhancedPAC()
    return (ConstPACField()
    return boolean IMPLEMENTATION_DEFINED "Bit 55 determines the size of the PAC field"; HavePACExt()
    && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
```

### Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2HaveEnhancedPAC

```
// HaveEnhancedPAC2()
// =====
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.
// HaveEnhancedPAC()
// =====
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    returnHaveEnhancedPAC()
    return ( HasArchVersionHavePACExt(ARMv8p6) || (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
```

### Library pseudocode for aarch64/functions/pac/pac/HaveFPACHaveEnhancedPAC2

```
// HaveFPAC()
// =====
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.
// HaveEnhancedPAC2()
// =====
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveFPAC()
HaveEnhancedPAC2()
    return (ARMv8p6) || (HasArchVersion(ARMv8p3HaveEnhancedPAC2HasArchVersion()) && boolean IMPLEMENTATION_DEFINED)
```

### Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombinedHaveFPAC

```
// HaveFPACCombined()
// =====
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.
// HaveFPAC()
// =====
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPACCombined()
HaveFPAC()
    return HaveFPACHaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality"
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACEExtHaveFPACCombined

```
// HavePACEExt()
// =====
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.
// HaveFPACCombined()
// =====
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HavePACEExt()
HaveFPACCombined()
    return HasArchVersionHaveFPAC(ARMv8p3);() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality"
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACIMPHavePACEExt

```
// HavePACIMP()
// HavePACEExt()
// =====
// Returns TRUE if support for PAC IMP is implemented, FALSE otherwise.
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACIMP()
HavePACEExt()
    return (ARMv8p3HavePACEExtHasArchVersion()) && boolean IMPLEMENTATION_DEFINED "Has PAC IMP functionality"
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3HavePACIMP

```
// HavePACQARMA3()
// =====
// Returns TRUE if support for PAC QARMA3 is implemented, FALSE otherwise.
// HavePACIMP()
// =====
// Returns TRUE if support for PAC IMP is implemented, FALSE otherwise.

boolean HavePACQARMA3()
HavePACIMP()
    return HavePACEExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 functionality";() && boolean IMPLEMENTATION_DEFINED
```

## Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5HavePACQARMA3

```
// HavePACQARMA5()
// HavePACQARMA3()
// =====
// Returns TRUE if support for PAC QARMA5 is implemented, FALSE otherwise.
// Returns TRUE if support for PAC QARMA3 is implemented, FALSE otherwise.

boolean HavePACQARMA5()
HavePACQARMA3()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 functionality";() && boolean IM
```

## Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRangesHavePACQARMA5

```
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.
// HavePACQARMA5()
// =====
// Returns TRUE if support for PAC QARMA5 is implemented, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime =HavePACQARMA5()
    return TranslationRegimeHavePACExt(PSTATE.EL, AccType_NORMAL);
    return HasUnprivileged(regime);() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 functionality";
```

## Library pseudocode for aarch64/functions/pac/strippac/StripPtrHasUpperAndLowerAddRanges

```
// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

bits(64)boolean Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi =PtrHasUpperAndLowerAddRanges()
    regime = EffectiveTBITranslationRegime(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit =(PSTATE.EL, CalculateBottomPACBitAccType_NORMAL(A<55>);
    extfield =);
    return ReplicateHasUnprivileged(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;

    return original_ptr;(regime);
```

## Library pseudocode for aarch64/functions/pac/trappacusestrip/TrapPACUseStrip

```
// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64)

TrapPACUse(bits(2) target_el)
    assertStrip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = HaveELEffectiveTBI(target_el) && target_el != (A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = EL0CalculateBottomPACBit && (A<55>);
    extfield = UIntReplicate(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset); (A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

```

// AArch64.ESB0peration()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

AArch64.ESB0peration()
    boolean mask_active;

    route_to_el3 = TrapPACUse(bits(2) target_el)
    assert HaveEL((target_el) && target_el != EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, && EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
        PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < (target_el) ->= UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);
(PSTATE.EL);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && bits(64) preferred_exception_return = IsSynchronizablePhysicalErrorPendingThisInstr/
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if(); ELUsingAArch32ExceptionRecord(exception;
    vect_offset = 0;
    exception = S1TranslationRegimeExceptionSyndrome() then
        syndrome32 = ( AArch32.PhysicalErrorSyndromeException_PACTrap();
        DISR = ); AArch32.ReportDeferredSErrorsAArch64.TakeException(syndrome32.AET, syndrome32.ExT);
    else
        implicit_esb = FALSE;
        syndrome64 = AArch64.PhysicalErrorSyndrome(implicit_esb);
        DISR_EL1 = AArch64.ReportDeferredSErrors(syndrome64);
        ClearPendingPhysicalError(); // Set ISR_EL1.A to 0

    return;(target_el, exception, preferred_exception_return, vect_offset);

```

## Library pseudocode for aarch64/functions/ ras/AArch64.PhysicalSErrorSyndromeAArch64.ESBOperation

```
// Return the SError syndrome
bits(25)// AArch64.ESB0peration()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64 AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);AArch64.ESB0peration()
boolean mask_active;

route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
route_to_el2 = (EL2Enabled() &&
                (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

if target == EL1 then
    mask_active = PSTATE.EL IN {EL0, EL1};
elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
    mask_active = PSTATE.EL IN {EL0, EL2};
else
    mask_active = PSTATE.EL == target;

mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
                                PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
intdis = Halted() || ExternalDebugInterruptsDisabled(target);
masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending() then
    // This function might be called for an interworking case, and INTdis is masking
    // the SError interrupt.
    if ELUsingAArch32(S1TranslationRegime()) then
        syndrome32 = AArch32.PhysicalSErrorSyndrome();
        DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    else
        implicit_esb = FALSE;
        syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
        DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
        ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0

return;
```

## Library pseudocode for aarch64/functions/ ras/AArch64.ReportDeferredSErrorAArch64.PhysicalSErrorSyndrome

```
// AArch64.ReportDeferredSError()
// =====
// Generate deferred SError syndrome

bits(64)// Return the SError syndrome
bits(25) AArch64.ReportDeferredSError(bits(25) syndrome)
bits(64) target;
target<31> = '1'; // A
target<24> = syndrome<24>; // IDS
target<23:0> = syndrome<23:0>; // ISS
return target;AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

## Library pseudocode for aarch64/functions/ ras/AArch64.vESBOperationAArch64.ReportDeferredSError

```
// AArch64.vESBOperation()
// =====
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state// AArch64.ReportDe
// =====
// Generate deferred SError syndrome
bits(64)

AArch64.vESBOperation()
    assert PSTATE.EL IN {AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1'; // A
    target<24> = syndrome<24>; // IDS
    target<23:0> = syndrome<23:0>; // ISS
    return target;EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
            HCR_EL2.VSE = '0'; // Clear pending virtual SError

    return;
```

## Library pseudocode for aarch64/

functions/**registers**/**ras**/**AArch64.MaybeZeroRegisterUppers****AArch64.vESBOperation**

```
// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero. // AArch64.vESBOperation
// =====
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.MaybeZeroRegisterUppers()
    assertAArch64.vESBOperation()
    assert PSTATE.EL IN { UsingAArch32(); // Always called from AArch32 state before entering AA

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elsif PSTATE.EL IN {} && EL0EL2Enabled,();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1) &&;
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if EL2Enabled() && !ELUsingAArch32(EL2EL1) then
            first = 0; last = 30; include_R15 = FALSE;
        else
            first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && VDISR = ConstrainUnpredictableBoolAArch32.ReportDeferred
        else
            VDISR_EL2 =Unpredictable_ZEROUPPERAArch64.ReportDeferredError) then
                R[n]<63:32> = Zeros();
    (VSESR_EL2<24:0>);
    HCR_EL2.VSE = '0'; // Clear pending virtual SError

return;
```

## Library pseudocode for aarch64/functions/ registers/**AArch64.ResetGeneralRegisters****AArch64.MaybeZeroRegisterUppers**

```
// AArch64.ResetGeneralRegisters()
// =====// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.ResetGeneralRegisters()

    for i = 0 to 30AArch64.MaybeZeroRegisterUppers()
    assert
        (); // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            R[n]<63:32> = ZerosXUsingAArch32[i] = bits(64) UNKNOWN;
    ());

    return;
```

## Library pseudocode for aarch64/functions/ registers/**AArch64.ResetSIMDFPRegisters****AArch64.ResetGeneralRegisters**

```
// AArch64.ResetSIMDFPRegisters()
// =====// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetSIMDFPRegisters()
AArch64.ResetGeneralRegisters()

    for i = 0 to 31 for i = 0 to 30
        VX[i] = bits(128) UNKNOWN;
    [i] = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/ registers/**AArch64.ResetSpecialRegisters****AArch64.ResetSIMDFPRegisters**

```
// AArch64.ResetSpecialRegisters()
// =====// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSpecialRegisters()
AArch64.ResetSIMDFPRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if for i = 0 to 31 HaveELV(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;
    [i] = bits(128) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/**AArch64.ResetSystemRegisters****AArch64.ResetSpecialRegisters**

```
// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1AArch64.ResetSystemRegisters(boolean cold_reset);) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/**PCAArch64.ResetSystemRegisters**

```
// PC - non-assignment form
// =====
// Read program counter.

bits(64) PC[]
return _PC;AArch64.ResetSystemRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/functions/registers/**SPPC**

```
// SP[] - assignment form
// =====
// Write to stack pointer from either a 32-bit or a 64-bit value.// PC - non-assignment form
// =====
// Read program counter.
bits(64)

SP[] = bits(width) value
  assert width IN {32,64};
  if PSTATE.SP == '0' then
    SP_EL0 = PC[]
  return _PC; ZeroExtend(value);
  else
    case PSTATE.EL of
      when EL0 SP_EL0 = ZeroExtend(value);
      when EL1 SP_EL1 = ZeroExtend(value);
      when EL2 SP_EL2 = ZeroExtend(value);
      when EL3 SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// =====
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
  assert width IN {8,16,32,64};
  if PSTATE.SP == '0' then
    return SP_EL0<width-1:0>;
  else
    case PSTATE.EL of
      when EL0 return SP_EL0<width-1:0>;
      when EL1 return SP_EL1<width-1:0>;
      when EL2 return SP_EL2<width-1:0>;
      when EL3 return SP_EL3<width-1:0>;
```

## Library pseudocode for aarch64/functions/registers/VSP

```
// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.
// SP[] - assignment form
// =====
// Write to stack pointer from either a 32-bit or a 64-bit value.

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    _V[n] = SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    return;

// V[] - non-assignment form
// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width)
    else
        case PSTATE.EL of
            when EL0 SP_EL0 = ZeroExtend(value);
            when EL1 SP_EL1 = ZeroExtend(value);
            when EL2 SP_EL2 = ZeroExtend(value);
            when EL3 SP_EL3 = ZeroExtend(value);
        return;

// SP[] - non-assignment form
// =====
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0 return SP_EL0<width-1:0>;
            when EL1 return SP_EL1<width-1:0>;
            when EL2 return SP_EL2<width-1:0>;
            when EL3 V[integer n]
                assert n >= 0 && n <= 31;
                assert width IN {8,16,32,64,128};
                return _V[n]<width-1:0>;
            return SP_EL3<width-1:0>;
```

## Library pseudocode for aarch64/functions/registers/VpartV

```
// Vpart[] - non-assignment form
// =====
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width)// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits. Vpart[integer n, integer part]
V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width IN {32,64};
        return _V[n]<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register. assert width IN {8,16,32,64,128};
_V[n] =
ZeroExtend(value);
return;

// V[] - non-assignment form
// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<(width * 2)-1:width> = value<width-1:0>;V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _V[n]<width-1:0>;
```



```

// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value.// Vpart[] - non-assignment form
// =====
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width)

X[integer n] = bits(width) value
Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width IN {32,64};
        return _V[n]<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register. ZeroExtend(value);
return;

// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X[integer n, integer width] = bits(width) value
Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);

// X[] - non-assignment form
// =====
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);(value);

```

```

else
    assert width == 64;
    _V[n]<(width * 2)-1:width> = value<width-1:0>;

```

## Library pseudocode for aarch64/functions/sysregisters/registers/CNTKCTLX

```

// CNTKCTL[] - non-assignment form
// =====

CNTKCTLType// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value. CNTKCTL[]
    bits(64) r;
    if X[integer n] = bits(width) value
        assert n >= 0 && n <= 31;
        assert width IN {32,64};
        if n != 31 then
            _R[n] = (value);
        return;

// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);

// X[] - non-assignment form
// =====
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return ZerosIsInHostZeroExtend() then
            r = CNTHCTL_EL2;
            return r;
        r = CNTKCTL_EL1;
        return r;(width);

```

## Library pseudocode for aarch64/functions/sysregisters/CNTKCTLTypeCNTKCTL

```
type// CNTKCTL[] - non-assignment form
// =====
CNTKCTLType CNTKCTLType;CNTKCTL[]
bits(64) r;
ifIsInHost() then
    r = CNTKCTL_EL2;
    return r;
r = CNTKCTL_EL1;
return r;
```

## Library pseudocode for aarch64/functions/sysregisters/CPACRCNTKCTLType

```
// CPACR[] - non-assignment form
// =====
CPACRType CPACR[]
bits(64) r;
ifCNTKCTLType; IsInHost() then
    r = CPTR_EL2;
    return r;
r = CPACR_EL1;
return r;
```

## Library pseudocode for aarch64/functions/sysregisters/CPACRTypeCPACR

```
type// CPACR[] - non-assignment form
// =====
CPACRType CPACRType;CPACR[]
bits(64) r;
ifIsInHost() then
    r = CPTR_EL2;
    return r;
r = CPACR_EL1;
return r;
```

## Library pseudocode for aarch64/functions/sysregisters/**ELRCPACRType**

```
// ELR[] - non-assignment form
// =====

bits(64) type ELR[bits(2) el]
  bits(64) r;
  case el of
    when CPACRType; EL1 r = ELR_EL1;
    when EL2 r = ELR_EL2;
    when EL3 r = ELR_EL3;
    otherwise Unreachable();
  return r;

// ELR[] - non-assignment form
// =====

bits(64) ELR[]
  assert PSTATE.EL != EL0;
  return ELR[PSTATE.EL];

// ELR[] - assignment form
// =====

ELR[bits(2) el] = bits(64) value
  bits(64) r = value;
  case el of
    when EL1 ELR_EL1 = r;
    when EL2 ELR_EL2 = r;
    when EL3 ELR_EL3 = r;
    otherwise Unreachable();
  return;

// ELR[] - assignment form
// =====

ELR[] = bits(64) value
  assert PSTATE.EL != EL0;
  ELR[PSTATE.EL] = value;
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ESRELR

```
// ESR[] - non-assignment form
// ELR[] - non-assignment form
// =====

ESRType bits(64) ESR[bits(2) regime]
ELR[bits(2) el]
    bits(64) r;
    case regime of
    case el of
        when EL1 r = ESR_EL1;
r = ELR_EL1;
        when EL2 r = ESR_EL2;
r = ELR_EL2;
        when EL3 r = ESR_EL3;
r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ELR[] - non-assignment form
// =====

ESRType bits(64) ESR[]
    return ELR[]
    assert PSTATE.EL != ESRELR0;
    return S1TranslationRegimeELR();
[PSTATE.EL];

// ESR[] - assignment form
// ELR[] - assignment form
// =====

ESR[bits(2) regime] = ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when ESRType value
            bits(64) r = value;
    case regime of
        when EL1 ESR_EL1 = r;
ELR_EL1 = r;
        when EL2 ESR_EL2 = r;
ELR_EL2 = r;
        when EL3 ESR_EL3 = r;
ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// ELR[] - assignment form
// =====

ESR[] = ELR[] = bits(64) value
    assert PSTATE.EL != ESRTypeEL0 value;
    ESRELR[S1TranslationRegime()] = value; [PSTATE.EL] = value;
    return;
```

```

type// ESR[] - non-assignment form
// =====

ESRType ESRType; ESR[bits(2) regime]
bits(64) r;
case regime of
    when EL1 r = ESR_EL1;
    when EL2 r = ESR_EL2;
    when EL3 r = ESR_EL3;
    otherwise Unreachable();
return r;

// ESR[] - non-assignment form
// =====

ESRType ESR[]
return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =====

ESR[bits(2) regime] = ESRType value
bits(64) r = value;
case regime of
    when EL1 ESR_EL1 = r;
    when EL2 ESR_EL2 = r;
    when EL3 ESR_EL3 = r;
    otherwise Unreachable();
return;

// ESR[] - assignment form
// =====

ESR[] = ESRType value
ESR[S1TranslationRegime()] = value;

```

```

// FAR[] - non-assignment form
// =====

bits(64) type FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when ESRType; EL1 r = FAR_EL1;
        when EL2 r = FAR_EL2;
        when EL3 r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// =====

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =====

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1 FAR_EL1 = r;
        when EL2 FAR_EL2 = r;
        when EL3 FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =====

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;

```

## Library pseudocode for aarch64/functions/sysregisters/MAIRFAR

```
// MAIR[] - non-assignment form
// =====
// FAR[] - non-assignment form
// =====

MAIRType bits(64) MAIR[bits(2) regime]
FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = MAIR_EL1;
r = FAR_EL1;
        when EL2 r = MAIR_EL2;
r = FAR_EL2;
        when EL3 r = MAIR_EL3;
r = FAR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// =====
// FAR[] - non-assignment form
// =====

MAIRType bits(64) MAIR[]
FAR[]
    return [SITranslationRegime()];

// FAR[] - assignment form
// =====

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1 FAR_EL1 = r;
        when EL2 FAR_EL2 = r;
        when EL3 FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =====

FAR[] = bits(64) value
    FARMAIRFAR[SITranslationRegime()];{} = value;
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/MAIRTypeMAIR

```
type// MAIR[] - non-assignment form
// =====

MAIRType MAIRType;MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = MAIR_EL1;
        when EL2 r = MAIR_EL2;
        when EL3 r = MAIR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// =====

MAIRType MAIR[]
    return MAIR[SITranslationRegime()];
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLRMAIRType

```
// SCTLR[] - non-assignment form
// =====

SCTLRType type SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when MAIRType; EL1 r = SCTLR_EL1;
        when EL2 r = SCTLR_EL2;
        when EL3 r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLRTypeSCTLR

```
type// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLRType; SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = SCTLR_EL1;
        when EL2 r = SCTLR_EL2;
        when EL3 r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/sysregisters/VBAR SCTLRType

```
// VBAR[] - non-assignment form
// =====

bits(64) type VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when SCTLRType; EL1 r = VBAR_EL1;
        when EL2 r = VBAR_EL2;
        when EL3 r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// =====

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

## Library pseudocode for aarch64/

### functions/systemsysregisters/AArch64.AllocationTagAccessIsEnabledVBAR

```
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.
// VBAR[] - non-assignment form
// =====

booleanbits(64) AArch64.AllocationTagAccessIsEnabled(VBAR[bits(2) regime])
    bits(64) r;
    case regime of
        when AccType acctype)
            bits(2) el = AArch64.AccessUsesEL(acctype);

    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, r = VBAR_EL1;
        when EL2} then
            return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {r = VBAR_EL2;
        when EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
            return FALSE;
    elsif SCTL_EL3.ATA == '0' && el == EL3 then
            return FALSE;
    elsif SCTL_EL2.ATA == '0' && el == r = VBAR_EL3;
        otherwise EL2Unreachable then
            return FALSE;
    elsif SCTL_EL1.ATA == '0' && el == ();
    return r;

// VBAR[] - non-assignment form
// =====

bits(64) EL1 then
    return FALSE;
    elsif SCTL_EL2.ATA0 == '0' && el == VBAR[]
    return EL0VBAR && EL2EnabledS1TranslationRegime() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTL_EL1.ATA0 == '0' && el == EL0 && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;();
```

## Library pseudocode for aarch64/functions/

### system/AArch64.ChooseNonExcludedTagAArch64.AllocationTagAccessIsEnabled

```
// AArch64.ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

bits(4) boolean AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;

    if AArch64.AllocationTagAccessIsEnabled( IsOnesAccType(exclude) ) then
        return '0000';

    if offset == '0000' then
        while exclude < acctype)
            bits(2) el = UIntAArch64.AccessUsesEL(tag) > == '1' do
                tag = tag + '0001';
(acctype);

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude < if SCR_EL3.ATA == '0' && el IN {, EL1, EL2} then
            return FALSE;
        elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
            return FALSE;
        elsif SCTL_EL3.ATA == '0' && el == EL3 then
            return FALSE;
        elsif SCTL_EL2.ATA == '0' && el == EL2 then
            return FALSE;
        elsif SCTL_EL1.ATA == '0' && el == EL1 then
            return FALSE;
        elsif SCTL_EL2.ATA0 == '0' && el == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
            return FALSE;
        elsif SCTL_EL1.ATA0 == '0' && el == EL0 && !(EL2EnabledUIntEL0(tag) > == '1' do
            tag = tag + '0001';

    return tag;() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;
```

## Library pseudocode for aarch64/functions/

### system/AArch64.ExecutingBROrBLROrRetInstrAArch64.ChooseNonExcludedTag

```
// AArch64.ExecutingBROrBLROrRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].
// AArch64.ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

boolean bits(4) AArch64.ExecutingBROrBLROrRetInstr()
    if !AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
        bits(4) tag = tag_in;
        bits(4) offset = offset_in;

    if HaveBTIExtIs0nes() then return FALSE;
(exclude) then
    return '0000';

    instr = if offset == '0000' then
        while exclude<(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UIntThisInstrUInt();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE; (tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

## Library pseudocode for aarch64/functions/

### system/AArch64.ExecutingBTIInstrAArch64.ExecutingBROrBLROrRetInstr

```
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.
// AArch64.ExecutingBROrBLROrRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBTIInstr()
AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

**Library pseudocode for aarch64/functions/  
system/AArch64.ExecutingERETInstrAArch64.ExecutingBTIInstr**

```
// AArch64.ExecutingERETInstr()
// =====
// Returns TRUE if current instruction is ERET.
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingERETInstr()
    instr = AArch64.ExecutingBTIInstr();
    if ! HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000'; if instr<31:22> == '1101010100' && instr<21:12> ==
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

**Library pseudocode for aarch64/functions/  
system/AArch64.ImpDefSysInstrAArch64.ExecutingERETInstr**

```
// Execute an implementation-defined system instruction with write (source operand). // AArch64.Executing
// =====
// Returns TRUE if current instruction is ERET.

boolean
AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t); AArch64
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

**Library pseudocode for aarch64/functions/  
system/AArch64.ImpDefSysInstrWithResultAArch64.ImpDefSysInstr**

```
// Execute an implementation-defined system instruction with read (result operand). // Execute an impleme
AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2); AArch64.
```

**Library pseudocode for aarch64/functions/  
system/AArch64.ImpDefSysRegReadAArch64.ImpDefSysInstrWithResult**

```
// Read from an implementation-defined system register and write the contents of the register to X[t]. //
AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t); AArch64.
```

**Library pseudocode for aarch64/functions/  
system/AArch64.ImpDefSysRegWriteAArch64.ImpDefSysRegRead**

```
// Write to an implementation-defined system register. // Read from an implementation-defined system regi
AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t); AArch64.
```

**Library pseudocode for aarch64/functions/  
system/AArch64.NextRandomTagBitAArch64.ImpDefSysRegWrite**

```
// AArch64.NextRandomTagBit()
// =====
// Generate a random bit suitable for generating a random Allocation Tag.

bit // Write to an implementation-defined system register. AArch64.NextRandomTagBit()
    bits(16) lfsr = RGSr_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSr_EL1.SEED = top:lfsr<15:1>;
    return top; AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
```

**Library pseudocode for aarch64/functions/  
system/AArch64.RandomTagAArch64.NextRandomTagBit**

```
// AArch64.RandomTag()  
// =====  
// Generate a random Allocation Tag.  
// AArch64.NextRandomTagBit()  
// =====  
// Generate a random bit suitable for generating a random Allocation Tag.  
  
bits(4)bit AArch64.RandomTag()  
    bits(4) tag;  
    for i = 0 to 3  
        tag<i> =AArch64.NextRandomTagBit()  
    bits(16) lfsr = RGSr_EL1.SEED;  
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;  
    RGSr_EL1.SEED = top:lfsr<15:1>;  
    return top; AArch64.NextRandomTagBit();  
    return tag;
```

**Library pseudocode for aarch64/functions/system/AArch64.SysInstrAArch64.RandomTag**

```
// Execute a system instruction with write (source operand). // AArch64.RandomTag()  
// =====  
// Generate a random Allocation Tag.  
  
bits(4)  
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);AArch64.Ran  
    bits(4) tag;  
    for i = 0 to 3  
        tag<i> =AArch64.NextRandomTagBit();  
    return tag;
```

**Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResultAArch64.SysInstr**

```
// Execute a system instruction with read (result operand).  
// Writes the result of the instruction to X[t]. // Execute a system instruction with write (source operand)  
AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);AArch64.Sy
```

**Library pseudocode for aarch64/functions/  
system/AArch64.SysRegReadAArch64.SysInstrWithResult**

```
// Read from a system register and write the contents of the register to X[t]. // Execute a system instruction  
// Writes the result of the instruction to X[t].  
AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);AArch64.Sy
```

**Library pseudocode for aarch64/functions/system/AArch64.SysRegWriteAArch64.SysRegRead**

```
// Write to a system register. // Read from a system register and write the contents of the register to X[t].  
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);AArch64.Sy
```

**Library pseudocode for aarch64/functions/system/BTypeCompatibleAArch64.SysRegWrite**

```
boolean BTypeCompatible; // Write to a system register. AArch64.SysRegWrite(integer op0, integer op1, inte
```

## Library pseudocode for aarch64/functions/system/BTypeCompatible\_BTIBTypeCompatible

```
// BTypeCompatible_BTI
// =====
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible; BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

## Library pseudocode for aarch64/functions/system/BTypeCompatible\_PACIXSPBTypeCompatible\_BTI

```
// BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.
// BTypeCompatible_BTI
// =====
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == BTypeCompatible_BTI(bits(2) hintcode)
        case hintcode of
            when '00'
                return FALSE;
            when '01'
                return PSTATE.BTYPE != '11';
            when '10'
                return PSTATE.BTYPE != '10';
            when '11'
                return TRUE; ELO then 35 else 36;
        return SCTLRL[]<index> == '0';
    else
        return FALSE;
```

## Library pseudocode for aarch64/functions/system/BTypeNextBTypeCompatible\_PACIXSP

```
bits(2) BTypeNext; BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == ELO then 35 else 36;
        return SCTLRL[]<index> == '0';
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTagBTypeNext

```
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// may choose to behave the same as GCR_EL1.RRND=0.
bits(4)bits(2) BTypeNext; ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

### Library pseudocode for aarch64/functions/system/InGuardedPageChooseRandomNonExcludedTag

```
boolean InGuardedPage; // The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// may choose to behave the same as GCR_EL1.RRND=0.
bits(4)ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

### Library pseudocode for aarch64/functions/system/IsHCRXEL2EnabledInGuardedPage

```
// IsHCRXEL2Enabled()
// =====
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

booleanboolean InGuardedPage; IsHCRXEL2Enabled()
    assert(HaveFeatHGX());
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

### Library pseudocode for aarch64/functions/system/SetBTypeCompatibleIsHCRXEL2Enabled

```
// SetBTypeCompatible()
// =====
// Sets the value of BTypeCompatible global variable used by BTI// IsHCRXEL2Enabled()
// =====
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean

SetBTypeCompatible(boolean x)
    BTypeCompatible = x; IsHCRXEL2Enabled()
    assert(HaveFeatHGX());
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

## Library pseudocode for aarch64/functions/system/**SetBTypeNext****SetBTypeCompatible**

```
// SetBTypeNext()  
// =====  
// Set the value of BTypeNext global variable used by BTI // SetBTypeCompatible()  
// =====  
// Sets the value of BTypeCompatible global variable used by BTI  
  
SetBTypeNext(bits(2) x)  
    BTypeNext = x; SetBTypeCompatible(boolean x)  
BTypeCompatible = x;
```

## Library pseudocode for aarch64/functions/system/**SetInGuardedPage****SetBTypeNext**

```
// SetInGuardedPage()  
// =====  
// Global state updated to denote if memory access is from a guarded page. // SetBTypeNext()  
// =====  
// Set the value of BTypeNext global variable used by BTI  
  
SetInGuardedPage(boolean guardedpage)  
    InGuardedPage = guardedpage; SetBTypeNext(bits(2) x)  
BTypeNext = x;
```

## Library pseudocode for aarch64/instrsfunctions/branchsystem/eret/ AArch64.ExceptionReturnSetInGuardedPage

```
// AArch64.ExceptionReturn()
// =====// SetInGuardedPage()
// =====
// Global state updated to denote if memory access is from a guarded page.

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if SetInGuardedPage(boolean guardedpage)
        InGuardedPage = guardedpage; HaveIESB() then
            sync_errors = SCTLRL[0].IESB == '1';
            if HaveDoubleFaultExt() then
                sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
            if sync_errors then
                SynchronizeErrors();
                iesb_req = TRUE;
                TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
            SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0'; // T32
        else
            new_pc<1:0> = '00'; // A32
        else // Return to AArch64
            // ELR_ELx[63:56] might include a tag
            new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

## Library pseudocode for aarch64/instrs/countopbranch/CountOperet/AArch64.ExceptionReturn

```

enumeration// AArch64.ExceptionReturn()
// ===== CountOp {AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if CountOp_CLZ,() then
        sync_errors = CountOp_CLS,[] .IESB == '1';
        if () then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalErrorInterrupts(iesb_req);
            SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elseif UsingAArch32() then // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0'; // T32
        else
            new_pc<1:0> = '00'; // A32
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatchCountOp_CNT};(FALSE); // Check for debug event on exception return

```

## Library pseudocode for aarch64/instrs/extendregcountop/DecodeRegExtendCountOp

```

// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendTypeenumeration DecodeRegExtend(bits(3) op)
    case op of
        when '000' return CountOp { ExtendType_UXTB;
        when '001' return CountOp_CLZ, ExtendType_UXTH;
        when '010' return CountOp_CLS, ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX; CountOp_CNT};

```

## Library pseudocode for aarch64/instrs/extendreg/ExtendRegDecodeRegExtend

```
// ExtendReg()
// =====
// Perform a register extension and shift
// DecodeRegExtend()
// =====
// Decode a register extension option

bits(N) ExtendType ExtendReg(integer reg, DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SCTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE; len = 8;
        when ;
        when '001' return ExtendType_UXTH unsigned = TRUE; len = 16;
        when ;
        when '010' return ExtendType_UXTW unsigned = TRUE; len = 32;
        when ;
        when '011' return ExtendType_UCTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = ;
    when '100' return MinExtendType_SXTB(len, N - shift);
    return ;
    when '101' return ExtendExtendType_SXTH(val<len-1:0> : ;
    when '110' return ;
    when '111' return ExtendType_SCTXZerosExtendType_SXTW(shift), N, unsigned); ;
```

## Library pseudocode for aarch64/instrs/extendreg/ExtendTypeExtendReg

```
enumeration// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendType {ExtendReg(integer reg,ExtendType_SXTB,exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = ExtendType_SXTH,[reg];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType_SXTW,unsigned = FALSE; len = 8;
        when ExtendType_SXTX,unsigned = FALSE; len = 16;
        when
            ExtendType_UXTB,unsigned = FALSE; len = 32;
        when ExtendType_UXTH,unsigned = FALSE; len = 64;
        when ExtendType_UXTW,unsigned = TRUE; len = 8;
        when unsigned = TRUE; len = 16;
        when ExtendType_UXTW unsigned = TRUE; len = 32;
        when ExtendType_UXTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : ZerosExtendType_UXTX);(shift), N, unsigned);
```

## Library pseudocode for aarch64/instrs/floatextendreg/arithmetic/max-min/fpmaxminop/FPMaxMinOpExtendType

```
enumeration FPMaxMinOp {ExtendType {FPMaxMinOp_MAX,ExtendType_SXTB, FPMaxMinOp_MIN,ExtendType_SXTH,
    FPMaxMinOp_MAXNUM,ExtendType_SXTW, FPMaxMinOp_MINNUM};ExtendType_SXTX,ExtendType
```

## Library pseudocode for aarch64/instrs/float/arithmetic/unarymax-min/fpunaryopfpmaxminop/FPUnaryOpFPMaxMinOp

```
enumeration FPUnaryOp {FPMaxMinOp {FPUnaryOp_ABS,FPMaxMinOp_MAX, FPUnaryOp_MOV,FPMaxMinOp_MIN,
    FPUnaryOp_NEG,FPMaxMinOp_MAXNUM, FPUnaryOp_SQRT};FPMaxMinOp_MINNUM};
```

## Library pseudocode for aarch64/instrs/float/convertarithmetic/fpconvopunary/FPConvOpfpunaryop/FPUnaryOp

```
enumeration FPConvOp {FPUnaryOp {FPConvOp_CVT_FtoI,FPUnaryOp_ABS, FPConvOp_CVT_ItoF,FPUnaryOp_MOV,
    FPConvOp_MOV_FtoI,FPUnaryOp_NEG, FPConvOp_MOV_ItoF
    ,FPUnaryOp_SQRT}; FPConvOp_CVT_FtoI_JS
};
```

## Library pseudocode for aarch64/

instrs/~~integer~~~~float~~/~~bitfield~~~~convert~~/~~bfxpreferred~~~~fpconvop~~/~~BFXPreferred~~~~FPConvOp~~

```
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean enumeration BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = FPConvOp { UInt(imms);
    integer R = FPConvOp_CVT_FtoI, UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if FPConvOp_CVT_ItoF, UInt(imms) < FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
        , UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE; FPConvOp_CVT_FtoI_JS
};
```



```

// DecodeBitMasks()
// =====
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIz, SBFIz, UXT[BH], SXT[BHW], LSL, LSR and ASR.

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M))boolean DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len =BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = HighestSetBitUInt(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels =(imms);
    integer R = ZeroExtendUInt((immr);

    // must not match UBFIz/SBFIz alias
    ifOnes(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    S = UInt(imms AND levels);
    R =(imms) < UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    // From a software perspective, the remaining code is equivalent to:
    //  esize = 1 << len;
    //  d = UInt(diff<len-1:0>);
    //  welem = ZeroExtend(Ones(S + 1), esize);
    //  telem = ZeroExtend(Ones(d + 1), esize);
    //  wmask = Replicate(ROR(welem, R));
    //  tmask = Replicate(telem);
    //  return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
        OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
        OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
        OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
        OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))

```

```

        OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
tmask = ((tmask
        AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
        OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
        AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
        OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
        AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
        OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
        AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
        OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
        AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
        OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
        AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
        OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
        AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
        OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));
(immr) then
    return FALSE;

if diff<6> != '0' then // borrow from S - R
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;
// must not match LSR/ASR/LSL alias (imms == 31 or 63)
if imms == sf:'11111' then
    return FALSE;

return (wmask<M-1:0>, tmask<M-1:0>); // must not match UXTx/SXTx alias
if immr == '000000' then
    // must not match 32-bit UXT[BH] or SXT[BH]
    if sf == '0' && imms IN {'000111', '001111'} then
        return FALSE;
    // must not match 64-bit SXT[BHW]
    if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
        return FALSE;

// must be UBFx/SBFx alias
return TRUE;

```



```

enumeration// DecodeBitMasks()
// =====

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) MoveWideOp {DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len =MoveWideOp_N,(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = MoveWideOp_Z,(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R; // 6-bit subtract with borrow

    // From a software perspective, the remaining code is equivalent to:
    // esize = 1 << len;
    // d = UInt(diff<len-1:0>);
    // welem = ZeroExtend(Ones(S + 1), esize);
    // telem = ZeroExtend(Ones(d + 1), esize);
    // wmask = Replicate(ROR(welem, R));
    // tmask = Replicate(telem);
    // return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
        OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
        OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
        OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
        OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
        OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
        OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask

```

```

        AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
        OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
        AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
        OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
        AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
        OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
        AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
        OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
        AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
        OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
        AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
        OR Replicate(Replicate(wmask_or<5>, 32) : ZerosMoveWideOp_K};(32), 1));

if diff<6> != '0' then // borrow from S - R
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<M-1:0>);

```

### Library pseudocode for aarch64/instrs/integer/logicalins- ext/movwpreferredinsert/MoveWidePreferredmovewide/movewideop/MoveWideOp

```

// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean enumeration MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = MoveWideOp_{ UInt(imms);
    integer R = MoveWideOp_N, MoveWideOp_Z, UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then
        return FALSE;
    if sf == '0' && !((immN:imms) IN {'00xxxxx'}) then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE; MoveWideOp_K};

```

## Library pseudocode for aarch64/instrs/integer/ShiftRegLogical/DecodeShiftMoveWidePreferred/MoveWidePreferred

```
// DecodeShift()
// =====
// Decode shift encodings
// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

ShiftType boolean DecodeShift(bits(2) op)
    case op of
        when '00' return MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = ShiftType_LSLUInt;
        when '01' return(imms);
    integer R = ShiftType_LSRUInt;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then
        return FALSE;
    if sf == '0' && !((immN:imms) IN {'00xxxxx'}) then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

## Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftRegDecodeShift

```
// ShiftReg()
// =====
// Perform shift of a register operand
// DecodeShift()
// =====
// Decode shift encodings

bits(N)ShiftType ShiftReg(integer reg,DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType shifttype, integer amount)
    bits(N) result = X[reg];
    case shifttype of
        when ShiftType_LSL result =;
        when '01' return LSL(result, amount);
        when ShiftType_LSR result =;
        when '10' return LSR(result, amount);
        when ShiftType_ASR result =;
        when '11' return ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;;
```

### Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftTypeShiftReg

```
enumeration// ShiftReg()  
// =====  
// Perform shift of a register operand  
  
bits(N) ShiftType {ShiftReg(integer reg, ShiftType_LSL, shiftype, integer amount)  
    bits(N) result = ShiftType_LSR, [reg];  
    case shiftype of  
        when ShiftType_ASR, result = (result, amount);  
        when ShiftType_LSR result = LSR(result, amount);  
        when ShiftType_ASR result = ASR(result, amount);  
        when ShiftType_ROR result = RORShiftType_ROR}; (result, amount);  
    return result;
```

### Library pseudocode for aarch64/instrs/logicalopinteger/LogicalOpshiftreg/ShiftType

```
enumeration LogicalOp {ShiftType {LogicalOp_AND, ShiftType_LSL, LogicalOp_EOR, ShiftType_LSR, LogicalOp_
```

### Library pseudocode for aarch64/instrs/memorylogicalop/memop/MemAtomicOpLogicalOp

```
enumeration MemAtomicOp {LogicalOp {MemAtomicOp_ADD, LogicalOp_AND,  
    MemAtomicOp_BIC, LogicalOp_EOR,  
    MemAtomicOp_EOR, LogicalOp_ORR};  
    MemAtomicOp_ORR,  
    MemAtomicOp_SMAX,  
    MemAtomicOp_SMIN,  
    MemAtomicOp_UMAX,  
    MemAtomicOp_UMIN,  
    MemAtomicOp_SWP};
```

### Library pseudocode for aarch64/instrs/memory/memop/MemOpMemAtomicOp

```
enumeration MemOp {MemAtomicOp {MemOp_LOAD, MemAtomicOp_ADD, MemOp_STORE, MemAtomicOp_BIC, MemOp_PREFETCH},  
    MemAtomicOp_SMAX,  
    MemAtomicOp_SMIN,  
    MemAtomicOp_UMAX,  
    MemAtomicOp_UMIN,  
    MemAtomicOp_SWP};
```

### Library pseudocode for aarch64/instrs/memory/prefetchmemop/PrefetchMemOp

```
// Prefetch()  
// =====  
  
// Decode and execute the prefetch hint on ADDRESS specified by PRFOPenumeration  
  
Prefetch(bits(64) address, bits(5) prfop)MemOp-{  
    PrefetchHint hint;  
    integer target;  
    boolean stream;  
  
    case prfop<4:3> of  
        when '00' hint = MemOp_LOAD, Prefetch_READ; // PLD: prefetch for load  
        when '01' hint = MemOp_STORE, Prefetch_EXEC; // PLI: preload instructions  
        when '10' hint = Prefetch_WRITE; // PST: prepare for store  
        when '11' return; // unallocated hint  
    target = UInt(prfop<2:1>); // target cache level  
    stream = (prfop<0> != '0'); // streaming (non-temporal)  
    Hint_Prefetch(address, hint, target, stream);  
    return; MemOp_PREFETCH};
```

## Library pseudocode for aarch64/instrs/systemmemory/barriers/prefetch/barrierop/ MemBarrierOpPrefetch

```

enumeration // Prefetch()
// =====
// Decode and execute the prefetch hint on ADDRESS specified by PRFOP MemBarrierOp {Prefetch(bits(64)-a
, hint;
integer target;
boolean stream;

case prfop<4:3> of
when '00' hint = MemBarrierOp_DMB // Data Memory Barrier
,; // PLD: prefetch for load
when '01' hint = MemBarrierOp_ISB // Instruction Synchronization Barrier
,; // PLI: preload instructions
when '10' hint = MemBarrierOp_SSBB // Speculative Synchronization Barrier to VA
,; // PST: prepare for store
when '11' return; // unallocated hint
target = MemBarrierOp_PSSBB // Speculative Synchronization Barrier to PA
, (prfop<2:1>); // target cache level
stream = (prfop<0> != '0'); // streaming (non-temporal) MemBarrierOp_SB // S
}; (address, hint, target, stream);

return;

```

## Library pseudocode for aarch64/instrs/ system/hintsbarriers/syshintopbarrierop/SystemHintOpMemBarrierOp

```

enumeration SystemHintOp {MemBarrierOp {
SystemHintOp_NOP, MemBarrierOp_DSB // Data Synchronization Barrier
,
SystemHintOp_YIELD, MemBarrierOp_DMB // Data Memory Barrier
,
SystemHintOp_WFE, MemBarrierOp_ISB // Instruction Synchronization Barrier
,
SystemHintOp_WFI, MemBarrierOp_SSBB // Speculative Synchronization Barrier to VA
,
SystemHintOp_SEV, MemBarrierOp_PSSBB // Speculative Synchronization Barrier to PA
,
SystemHintOp_SEVL, MemBarrierOp_SB // Speculation Barrier
};

SystemHintOp_DGH,
SystemHintOp_ESB,
SystemHintOp_PSB,
SystemHintOp_TSB,
SystemHintOp_BTI,
SystemHintOp_WFET,
SystemHintOp_WFIT,
SystemHintOp_CSDB
};

```

## Library pseudocode for aarch64/instrs/system/registerhints/cpsrsyshintop/pstatefield/ PSTATEFieldSystemHintOp

```

enumeration PSTATEField {SystemHintOp {PSTATEField_DAIFSet, SystemHintOp_NOP, PSTATEField_DAIFClr, SystemH
PSTATEField_PAN, // Armv8.1SystemHintOp_WFE,
PSTATEField_UAO, // Armv8.2SystemHintOp_WFI,
PSTATEField_DIT, // Armv8.4SystemHintOp_SEV,
PSTATEField_SSBS, SystemHintOp_SEVL,
PSTATEField_TCO, // Armv8.5SystemHintOp_DGH,
PSTATEField_ALLINT, SystemHintOp_ESB,
PSTATEField_SP
}; SystemHintOp_PSB, SystemHintOp_TSB,
SystemHintOp_BTI,
SystemHintOp_WFET,
SystemHintOp_WFIT,
SystemHintOp_CSDB
};

```



```

// AArch64.AT()
// =====
// Perform address translation as per AT instructions.enumeration

AArch64.AT(bits(64) address, PSTATEField { TranslationStage stage_in, bits(2) el_in, PSTATEField_DAIFSet,
TranslationStage stage = stage_in;
bits(2) el = el_in;
// For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
// the EL2&0 translation regime is used.
if HCR_EL2.<E2H, TGE> == '11' && el == PSTATEField_PAN, // Armv8.1 EL1 && stage == PSTATEField_UAO, //
el = PSTATEField_DIT, // Armv8.4 EL2;
if PSTATEField_SSBS, HaveEL(PSTATEField_TC0, // Armv8.5 EL3) && stage == PSTATEField_ALLINT, Translation
stage = TranslationStage_1;

acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;
iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
aligned = TRUE;
ispriv = el != EL0;

fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

Regime regime;
if stage == TranslationStage_12 then
    regime = Regime_EL10;
else
    regime = TranslationRegime(el, acctype);

AddressDescriptor addrdesc;
ss = SecurityStateAtEL(el);
if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, address<31:0>, acctype,
aligned, iswrite, ispriv);
    else
        (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, ss, address<31:0>, acctype,
aligned, iswrite, ispriv);
else
    (fault, addrdesc) = AArch64.S1Translate(fault, regime, ss, address, acctype, aligned,
iswrite, ispriv);

if stage == TranslationStage_12 && fault.statuscode == Fault_None then
    boolean s2fslwalk;
    boolean slaarch64;
    if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
        addrdesc.vaddress = ZeroExtend(address);
        s2fslwalk = FALSE;
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fslwalk, acctype,
aligned, iswrite, ispriv);
    elsif regime == Regime_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        s2fslwalk = FALSE;
        (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, slaarch64, ss, s2fslwalk,
acctype, aligned, iswrite, ispriv);

is_ATS1Ex = stage != TranslationStage_12;
if fault.statuscode != Fault_None then
    addrdesc = CreateFaultyAddressDescriptor(address, fault);
    // Take an exception on:
    // * A Synchronous External abort occurs on translation table walk
    // * A stage 2 fault occurs on a stage 1 walk
    if IsExternalAbort(fault) || (PSTATE.EL == EL1 && fault.s2fslwalk) then
        PAR_EL1 = bits(64) UNKNOWN;
        AArch64.Abort(address, addrdesc.fault);

AArch64.EncodePAR(regime, addrdesc);
return; PSTATEField_SP
};

```



```

// AArch64.EncodePAR()
// =====
// Encode PAR register with result of translation. // AArch64.AT()
// =====
// Perform address translation as per AT instructions.

AArch64.EncodePAR(AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess,
    TranslationStage stage = stage_in;
    bits(2) el = el_in;
    // For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage_1 then
        el = EL2;
    if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() then
        stage = TranslationStage_1;

    acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;
    iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    aligned = TRUE;
    ispriv = el != EL0;

    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    Regime regime, regime;
    if stage == TranslationStage_12 then
        regime = Regime_EL10;
    else
        regime = TranslationRegime(el, acctype);

    AddressDescriptor addrdesc)
    PAR_EL1 = addrdesc;
    ss = ZerosSecurityStateAtEL();
    paspace = addrdesc.paddress.paspace;

    if !(el);
    if (el == IsFaultEL0(addrdesc) then
        PAR_EL1.F = '0';
        PAR_EL1<11> = '1'; // RES1
        if && SecurityStateForRegimeELUsingAArch32(regime) == (SS_SecureEL1 then
            PAR_EL1.NS = if paspace ==)) || (el != PAS_SecureEL0 then '0' else '1';
        else
            PAR_EL1.NS = bit UNKNOWN;
        PAR_EL1.SH = ReportedPARShareability(&&PAREncodeShareabilityELUsingAArch32(addrdesc.memattrs));
        PAR_EL1.PA = addrdesc.paddress.address<52-1:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(el) then
            if regime == EncodePARAttrsRegime_EL2(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F = '1';
        PAR_EL1.FST = || TTBCR.EAE == '1' then
            (fault, addrdesc) = (fault, regime, ss, address<31:0>, acctype,
                aligned, iswrite, ispriv);
        else
            (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, ss, address<31:0>, acctype,
                aligned, iswrite, ispriv);
    else
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, ss, address, acctype, aligned,
            iswrite, ispriv);

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        boolean s2fslwalk;
        boolean slaarch64;
        if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address);
            s2fslwalk = FALSE;
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fslwalk, acctype,
                aligned, iswrite, ispriv);
        elsif regime == Regime_EL10 && EL2Enabled() then

```

```

        slaarch64 = TRUE;
        s2fslwalk = FALSE;
        (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, slaarch64, ss, s2fslwalk,
                                                acctype, aligned, iswrite, ispriv);

        is_ATS1Ex = stage != TranslationStage_12;
        if fault.statuscode != Fault_None then
            addrdesc = CreateFaultyAddressDescriptor(address, fault);
            // Take an exception on:
            // * A Synchronous external abort occurs on translation table walk
            // * A stage 2 fault occurs on a stage 1 walk
            if IsExternalAbort(fault) || (PSTATE.EL == EL1 && fault.s2fslwalk) then
                PAR_EL1 = bits(64) UNKNOWN;
                AArch64.Abort(address, addrdesc.fault);

        AArch64.EncodePARAArch64.PARFaultStatusAArch32.S1TranslateLD(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fslwalk then '1' else '0';
        PAR_EL1.S = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    (regime, addrdesc);
    return;

```

### Library pseudocode for aarch64/instrs/system/sysops/ at/AArch64.PARFaultStatusAArch64.EncodePAR

```

// AArch64.PARFaultStatus()
// =====
// Fault status field decoding of 64-bit PAR.

bits(6) // AArch64.EncodePAR()
// =====
// Encode PAR register with result of translation. AArch64.PARFaultStatus(AArch64.EncodePAR(FaultRecordRe

    bits(6) fst;

    if fault.statuscode == regime, Fault_DomainAddressDescriptor then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = addrdesc;
        PAR_EL1 = ();
        paspace = addrdesc.paddress.paspace;

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        PAR_EL1<11> = '1'; // RES1
        if SecurityStateForRegime(regime) == SS_Secure then
            PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
        else
            PAR_EL1.NS = bit UNKNOWN;
            PAR_EL1.SH = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
            PAR_EL1.PA = addrdesc.paddress.address<52-1:12>;
            PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
            PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F = '1';
        PAR_EL1.FST = AArch64.PARFaultStatusEncodeLDFSCZeros(fault.statuscode, fault.level);
    return fst; (addrdesc.fault);
    PAR_EL1.PTW = if addrdesc.fault.s2fslwalk then '1' else '0';
    PAR_EL1.S = if addrdesc.fault.secondstage then '1' else '0';
    PAR_EL1<11> = '1'; // RES1
    PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;

```



```

// AArch64.DC()
// =====
// Perform Data Cache Operation.// AArch64.PARFaultStatus()
// =====
// Fault status field decoding of 64-bit PAR.

bits(6)

AArch64.DC(bits(64) regval, AArch64.PARFaultStatus( CacheTypeFaultRecord cachetype, fault)
    bits(6) fst;

    if fault.statuscode == CacheOpFault_Domain cacheop, then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = CacheOpScopeEncodeLDFSC opscope_in;
    CacheOpScope opscope = opscope_in;
    AccType acctype = AccType_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    cache.cachetype = cachetype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;

    if opscope == CacheOpScope_SetWay then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            (HCR_EL2.SWI0 == '1' || HCR_EL2.<DC,VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid = ASID[];
    else
        cache.is_asid_valid = FALSE;

    if opscope == CacheOpScope_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP"
        opscope = CacheOpScope_PoP;
    if opscope == CacheOpScope_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP"
        opscope = CacheOpScope_PoC;
    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp_Invalidate;
    vaddress = regval;

    size = 0; // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;

```

```

cache.vaddress = vaddress;

if need_translate then
    wasaligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    memattrs = memaddrdesc.memattrs;
    cache.paddress = memaddrdesc.paddress;
    cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
    if opscope IN {CacheOpScope_PoC, CacheOpScope_PoP, CacheOpScope_PoDP} then
        cache.shareability = memattrs.shareability;
    else
        cache.shareability = Shareability_NSH;
else
    cache.shareability = Shareability_UNKNOWN;
    cache.paddress = FullAddress UNKNOWN;

if cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.<DC,VM> != '00' then
    cache.cacheop = CacheOp_CleanInvalidate;

CACHE_OP(cache);
return; (fault.statuscode, fault.level);
return fst;

```



```

// AArch64.MemZero()
// =====// AArch64.DC()
// =====
// Perform Data Cache Operation.

AArch64.MemZero(bits(64) regval, AArch64.DC(bits(64) regval, CacheType cachetype) cachetype,

    CacheOp cacheop, CacheOpScope opscope_in)
    CacheOpScope opscope = opscope_in;
    AccType acctype = AccType_DCZVAAccType_DC;
    boolean iswrite = TRUE;
    boolean wasaligned = TRUE;

    integer size = 4*(2^(;CacheRecord cache;

    cache.acctype = acctype;
    cache.cachetype = cachetype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;

    if opscope == CacheOpScope_SetWay then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            (HCR_EL2.SWI0 == '1' || HCR_EL2.<DC, VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid = ASID[];
    else
        cache.is_asid_valid = FALSE;

    if opscope == CacheOpScope_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP"
        opscope = CacheOpScope_PoP;
    if opscope == CacheOpScope_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP"
        opscope = CacheOpScope_PoC;
    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp_Invalidate;
    vaddress = regval;

    size = 0; // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(DCZID_EL0.BS)));
    bits(64) vaddress = (CTR_EL0.DminLine)) && size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
    vaddress = Align(regval, size);

    memaddrdesc = cache.translated = need_translate;
    cache.vaddress = vaddress;

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

```

```

if IsFault(memaddrdesc) then
    if(memaddrdesc) then IsDebugException(memaddrdesc.fault) then
        AArch64.Abort(vaddress, memaddrdesc.fault);
    else(regval, memaddrdesc.fault);

    memattrs = memaddrdesc.memattrs;
    cache.paddress = memaddrdesc.paddress;
    cache.cpas =
        AArch64.AbortCPASAtPAS(regval, memaddrdesc.fault);
else
    if cachetype ==(memaddrdesc.paddress.paspace);
    if opscope IN { CacheType_DataCacheOpScope_PoC then,
        AArch64.DataMemZeroCacheOpScope_PoP(regval, vaddress, memaddrdesc, size);
    elsif cachetype ==, CacheType_TagCacheOpScope_PoDP then
        if} then
            cache.shareability = memattrs.shareability;
        else
            cache.shareability = HaveMTEExtShareability_NSH() then;
        else
            cache.shareability = AArch64.TagMemZeroShareability(vaddress, size);
        elsif cachetype ==UNKNOWN;
        cache.paddress = CacheType_Data_TagFullAddress then
            ifUNKNOWN;

        if cacheop == HaveMTEExtCacheOp_Invalidate() then&& PSTATE.EL == AArch64.TagMemZeroEL1(vaddress, size
            () && HCR_EL2.<DC,VM> != '00' then
                cache.cacheop = CacheOp_CleanInvalidate;

        CACHE_OPAArch64.DataMemZeroEL2Enabled(regval, vaddress, memaddrdesc, size);
(cache);
return;

```



```

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.// AArch64.MemZero()
// =====

AArch64.IC(AArch64.MemZero(bits(64) regval, CacheOpScopeCacheType opscope)
    regval = bits(64) UNKNOWN; cachetype)
    AArch64.IC(regval, opscope);

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_ICAccType_DCZVA;
    boolean iswrite = TRUE;
    boolean wasaligned = TRUE;

    cache.acctype = acctype;
    cache.cachetype = integer size = 4*(2^( CacheType_InstructionUInt;
    cache.cacheop =(DCZID_EL0.BS)));
    bits(64) vaddress = CacheOp_InvalidateAlign;
    cache.opscope = opscope;
    (regval, size);

    if opscope IN { memaddrdesc =CacheOpScope_ALLUAArch64.TranslateAddress, (vaddress, acctype, iswrite

    if CacheOpScope_ALLUIIsFault} then
        ss =(memaddrdesc) then
            if SecurityStateAtELIsDebugException(PSTATE.EL);
            cache.cpas =(memaddrdesc.fault) then CPASatSecurityStateAArch64.Abort(ss);
            if (opscope ==(vaddress, memaddrdesc.fault));
            else CacheOpScope_ALLUISAArch64.Abort || (opscope ==(regval, memaddrdesc.fault));
        else
            if cachetype == CacheOpScope_ALLUCacheType_Data && PSTATE.EL ==then EL1AArch64.DataMemZero
                &&(regval, vaddress, memaddrdesc, size);
            elsif cachetype == EL2EnabledCacheType_Tag() && HCR_EL2.FB == '1') then
                cache.shareability =then
                if Shareability_ISHHaveMTEExt;
            else
                cache.shareability =() then Shareability_NSHAArch64.TagMemZero;
                cache.regval = regval; (vaddress, size);
            elsif cachetype ==
                CACHE_OP_CacheType_Data_Tag(cache);
        else
            assert opscope ==then
            if CacheOpScope_PoUHaveMTEExt;

            if() then EL2EnabledAArch64.TagMemZero() && !(vaddress, size); IsInHostAArch64.DataMemZero() then
                if PSTATE.EL IN {EL0, EL1} then
                    cache.is_vmid_valid = TRUE;
                    cache.vmid = VMID[];
                else
                    cache.is_vmid_valid = FALSE;
            else
                cache.is_vmid_valid = FALSE;

            if PSTATE.EL == EL0 then
                cache.is_asid_valid = TRUE;
                cache.asid = ASID[];
            else
                cache.is_asid_valid = FALSE;

            bits(64) vaddress = regval;
            need_translate = ICInstNeedsTranslation(opscope);

            cache.vaddress = regval;
            cache.shareability = Shareability_NSH;
            cache.translated = need_translate;

```

```

    if !need_translate then
        cache.paddress = FullAddress UNKNOWN;
        CACHE_OP(cache);
        return;
    iswrite = FALSE;
    wasaligned = TRUE;
    size = 0;
    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
    cache.paddress = memaddrdesc.paddress;
    CACHE_OP(cache);
    (regval, vaddress, memaddrdesc, size);
    return;

```



```

// RestrictPrediction()
// =====
// Clear all predictions in the context.// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.RestrictPrediction(bits(64) val, AArch64.IC( RestrictTypeCacheOpScope restriction) opscope)
    regval = bits(64) UNKNOWN;

    ExecutionCntxtAArch64.IC c;
    target_el = val<25:24>;
    (regval, opscope);

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if// AArch64.IC()
    // =====
    // Perform Instruction Cache Operation. UInt(target_el) > AArch64.IC(bits(64) regval, UIntCacheOpScope(PST

    bit ns = val<26>;
    ss = opscope) TargetSecurityStateCacheRecord(ns);

    c.security = ss;
    c.target_el = target_el;

    if cache; AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if (opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && !() && HCR_EL2.FB == '1')) then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                c.is_vmid_valid = TRUE;
                c.all_vmid = FALSE;
                c.vmid = cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
            else
                cache.is_vmid_valid = FALSE;

        elsif target_el IN { if PSTATE.EL == EL0, then
            cache.is_asid_valid = TRUE;
            cache.asid = EL1ASID} then
                c.is_vmid_valid = TRUE;
                c.all_vmid = val<48> == '1';
                c.vmid = val<47:32>; // Only valid if val<48> == '0';
    [];

    else
        c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;
        cache.is_asid_valid = FALSE;

    if PSTATE.EL == bits(64) vaddress = regval;

```

```

need_translate = EL0ICInstNeedsTranslation then
    c.is_asid_valid = TRUE;
    c.all_asid      = FALSE;
    c.asid          = (opscope);
;
    cache.vaddress = regval;
    cache.shareability = ASIDShareability_NSH[];
;
    cache.translated = need_translate;

elseif target_el ==      if !need_translate then
    cache.paddress = EL0FullAddress then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<16> == '1';
        c.asid          = val<15:0>; // Only valid if val<16> == '0';
    else
        c.is_asid_valid = FALSE;
    c.restriction = restriction;UNKNOWN;
    (cache);
    return;
    iswrite = FALSE;
    wasaligned = TRUE;
    size = 0;
    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    cache.cpas = CPASatPAS(memaddrdesc.paddress.paspace);
    cache.paddress = memaddrdesc.paddress;
    CACHE_OPRESTRICT_PREDICTIONS CACHE_OP(c); (cache);
return;

```

Library pseudocode for aarch64/instrs/system/  
sysops/~~sysoppredictionrestrict~~/SysOpRestrictPrediction

```

// SysOp()
// =====

SystemOp// RestrictPrediction()
// =====
// Clear all predictions in the context. SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return AArch64.RestrictPrediction(bits(64) val, Sys_ATRestrictType; //
        when '100 0111 1000 000' return restriction; Sys_ATEXecutionCntxt; // S1E2R
        when '110 0111 1000 000' return c;
    target_el = val<25:24>;

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if Sys_ATUInt; // S1E3R
        when '000 0111 1000 001' return (target_el) > Sys_ATUInt; // S1E1W
        when '100 0111 1000 001' return (PSTATE.EL) then return;

    bit ns = val<26>;
    ss = Sys_ATTARGETSecurityState; // S1E2W
        when '110 0111 1000 001' return (ns);

    c.security = ss;
    c.target_el = target_el;

    if Sys_ATEL2Enabled; // S1E3W
        when '000 0111 1000 010' return () && ! Sys_ATIsInHost; // S1E0R
        when '000 0111 1000 011' return () then
            if PSTATE.EL IN { Sys_ATEL0; // S1E0W
                when '100 0111 1000 100' return; Sys_ATEL1; // S1E1R
                when '100 0111 1000 101' return; then
                    c.is_vmid_valid = TRUE;
                    c.all_vmid = FALSE;
                    c.vmid = Sys_ATVMID; // S1E1W
                when '100 0111 1000 110' return [];

            elsif target_el IN { Sys_ATEL0; // S1E0R
                when '100 0111 1000 111' return; Sys_ATEL1; // S1E0W
                when '011 0111 0100 001' return; then
                    c.is_vmid_valid = TRUE;
                    c.all_vmid = val<48> == '1';
                    c.vmid = val<47:32>; // Only valid if val<48> == '0';
                else
                    c.is_vmid_valid = FALSE;
            else
                c.is_vmid_valid = FALSE;

            if PSTATE.EL == Sys_DCEL0; // ZVA
                when '000 0111 0110 001' return; then
                    c.is_asid_valid = TRUE;
                    c.all_asid = FALSE;
                    c.asid = Sys_DCASID; // IVAC
                when '000 0111 0110 010' return [];

            elsif target_el == Sys_DCEL0; // ISW
                when '011 0111 1010 001' return; then
                    c.is_asid_valid = TRUE;
                    c.all_asid = val<16> == '1';
                    c.asid = val<15:0>; // Only valid if val<16> == '0';
            else
                c.is_asid_valid = FALSE;

            c.restriction = restriction; Sys_DCRESTRICT_PREDICTIONS; // CVAC
                when '000 0111 1010 010' return Sys_DC; // CSW
                when '011 0111 1011 001' return Sys_DC; // CVAU
                when '011 0111 1110 001' return Sys_DC; // CIVAC
                when '000 0111 1110 010' return Sys_DC; // CISW
                when '011 0111 1101 001' return Sys_DC; // CVADP
                when '000 0111 0001 000' return Sys_IC; // IALLUIS

```

```

when '000 0111 0101 000' return Sys_IC; // IALLU
when '011 0111 0101 001' return Sys_IC; // IVAU
when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
when '100 1000 0111 000' return Sys_TLBI; // ALLE2
when '110 1000 0111 000' return Sys_TLBI; // ALLE3
when '000 1000 0111 001' return Sys_TLBI; // VAE1
when '100 1000 0111 001' return Sys_TLBI; // VAE2
when '110 1000 0111 001' return Sys_TLBI; // VAE3
when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
when '000 1000 0111 011' return Sys_TLBI; // VAAE1
when '100 1000 0111 100' return Sys_TLBI; // ALLE1
when '000 1000 0111 101' return Sys_TLBI; // VALE1
when '100 1000 0111 101' return Sys_TLBI; // VALE2
when '110 1000 0111 101' return Sys_TLBI; // VALE3
when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
when '000 1000 0111 111' return Sys_TLBI; // VAALE1
return Sys_SYS; (c);

```

## Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOpSysOp

```

enumeration// SysOp()
// =====

SystemOp SystemOp {SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  case op1:CRn:CRm:op2 of
    when '000 0111 1000 000' return Sys_AT; // S1E1R
    when '100 0111 1000 000' return Sys_DC; // S1E2R
    when '110 0111 1000 000' return Sys_IC; // S1E3R
    when '000 0111 1000 001' return Sys_TLBI; // S1E1W
    when '100 0111 1000 001' return ; // S1E2W
    when '110 0111 1000 001' return Sys_AT; // S1E3W
    when '000 0111 1000 010' return Sys_AT; // S1E0R
    when '000 0111 1000 011' return Sys_AT; // S1E0W
    when '100 0111 1000 100' return Sys_AT; // S12E1R
    when '100 0111 1000 101' return Sys_AT; // S12E1W
    when '100 0111 1000 110' return Sys_AT; // S12E0R
    when '100 0111 1000 111' return Sys_AT; // S12E0W
    when '011 0111 0100 001' return Sys_DC; // ZVA
    when '000 0111 0110 001' return Sys_DC; // IVAC
    when '000 0111 0110 010' return Sys_DC; // ISW
    when '011 0111 1010 001' return Sys_DC; // CVAC
    when '000 0111 1010 010' return Sys_DC; // CSW
    when '011 0111 1011 001' return Sys_DC; // CVAU
    when '011 0111 1110 001' return Sys_DC; // CIVAC
    when '000 0111 1110 010' return Sys_DC; // CISW
    when '011 0111 1101 001' return Sys_DC; // CVADP
    when '000 0111 0001 000' return Sys_IC; // IALLUIS
    when '000 0111 0101 000' return Sys_IC; // IALLU
    when '011 0111 0101 001' return Sys_IC; // IVAU
    when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
    when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
    when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
    when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
    when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
    when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
    when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
    when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
    when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
    when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
    when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
    when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
    when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
    when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
    when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
    when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
    when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
    when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
    when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
    when '100 1000 0111 000' return Sys_TLBI; // ALLE2
    when '110 1000 0111 000' return Sys_TLBI; // ALLE3
    when '000 1000 0111 001' return Sys_TLBI; // VAE1
    when '100 1000 0111 001' return Sys_TLBI; // VAE2
    when '110 1000 0111 001' return Sys_TLBI; // VAE3
    when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
    when '000 1000 0111 011' return Sys_TLBI; // VAAE1
    when '100 1000 0111 100' return Sys_TLBI; // ALLE1
    when '000 1000 0111 101' return Sys_TLBI; // VALE1
    when '100 1000 0111 101' return Sys_TLBI; // VALE2
    when '110 1000 0111 101' return Sys_TLBI; // VALE3
    when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
    when '000 1000 0111 111' return Sys_TLBI; // VAALE1
  return Sys_SYSSys_SYS;;

```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/sysop/AArch32.DTLBI\_ALLSystemOp

```
// AArch32.DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.enumeration

AArch32.DTLBI_ALL(SystemOp {SecurityState security, Sys_AT, Regime regime, Sys_DC, Shareability shareability,
    assert PSTATE.EL IN {Sys_TLBI, EL3, EL2, EL1};

    TLBIRecord r;
    r.op      = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime   = regime;
    r.level    = TLBILevel_Any;
    r.attr     = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;Sys_SYS};
```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI\_ASIDAArch32.DTLBI\_ALL

```
// AArch32.DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// AArch32.DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.// order to deem this operation to be completed.

AArch32.DTLBI_ASID(AArch32.DTLBI_ALL(SecurityState security, Regime regime, bits(16) vmid, regime, Shareability shareability,
    TLBIMemAttr attr, bits(32) Rt)
attr);
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op      = TLBIOp_DASIDTLBIOp_DALL;
r.from_aarch64 = FALSE;
r.security = security;
r.regime   = regime;
r.vmid     = vmid;
r.level    = TLBILevel_Any;
r.attr     = attr;
r.asid     = Zeros(8) : Rt<7:0>;
r.attr     = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
```

```

// AArch32.DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// AArch32.DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//   TLBILevel_Any : this applies to TLB entries at all levels
//   TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_VA(AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                                     Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
  assert PSTATE.EL IN {EL3, EL2, EL1};

  TLBIRecord r;
  r.op = TLBIOp_DVATLBIOp_DASID;
  r.from_aarch64 = FALSE;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = level;
  r.attr = attr;
  r.asid = r.level = ZerosTLBILevel_Any(8) : Rt<7:0>;
  r.address = Zeros(32) : Rt<31:12> ;
  r.attr = attr;
  r.asid = Zeros(12);(8) : Rt<7:0>;

  TLBI(r);
  if shareability != Shareability_NSH then Broadcast(shareability, r);
  return;

```

```

// AArch32.ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// AArch32.DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_ALL(AArch32.DTLBI_VA(SecurityState security, Regime regime, regime, bits(16) vmid, Shareability
attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_IALLTLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = (8) : Rt<7:0>;
    r.address = Zeros(32) : Rt<31:12> : ZerosTLBILevel_AnyZeros;
    r.attr = attr;(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// AArch32.ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.// order to deem this operation to be completed.

AArch32.ITLBI_ASID(AArch32.ITLBI_ALL(SecurityState security, Regime regime, bits(16) vmid, regime, Shareability shareability,
                                     TLBMemAttr attr, bits(32) Rt)
attr);
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_IASIDTLBIOp_IALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = TLBILevel_Any;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.ITLBI_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// AArch32.ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//   TLBILevel_Any : this applies to TLB entries at all levels
//   TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_VA(AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_IVATLBIOp_IASID;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = r.level = ZerosTLBILevel_Any(8) : Rt<7:0>;
r.address = Zeros(32) : Rt<31:12> :;
r.attr = attr;
r.asid = Zeros(12):(8) : Rt<7:0>;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

```

```

// AArch32.TLBI_ALL()
// AArch32.ITLBI_VA()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ALL(AArch32.ITLBI_VA(SecurityState security, Regime regime, regime, bits(16) vmid, Shareability
attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};

    EL1};

    TLBIRecord r;
    r.op = TLBIOp_ALLTLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = (8) : Rt<7:0>;
    r.address = Zeros(32) : Rt<31:12> : ZerosTLBILevel_AnyZeros;
    r.attr = attr; (12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// AArch32.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ASID(AArch32.TLBI_ALL(SecurityState security, Regime regime, bits(16) vmid,regime, Shareabi
    TLBIMemAttr attr, bits(32) Rt)
attr);
    assert PSTATE.EL IN {EL3, EL2,}; EL1};

    TLBIRecord r;
    r.op          = TLBIOp_ASIDTLBIOp_ALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.attr        = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// AArch32.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_IPAS2(AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                                     Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security ==, SS_NonSecureEL1;

    TLBIRecord r;
    r.op          = TLBIOp_IPAS2TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = r.level = ZerosTLBILevel_Any(24) : Rt<27:0> :;
    r.attr         = attr;
    r.asid         = Zeros(12);
    r.ipaspace     = PAS_NonSecure;(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// AArch32.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VA(AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2,};
assert security == EL1SS_NonSecure;;

    TLBIRecord r;
    r.op          = TLBIOp_VATLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = r.address = Zeros(8) : Rt<7:0>;
    r.address      = (24) : Rt<27:0> : Zeros(32) : Rt<31:12> : (12);
    r.ipaspace     = ZerosPAS_NonSecure(12);;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.TLBI_VAA()
// =====
// AArch32.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VAA(AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = ;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = Zeros(TLBIOp_VAATLBIOp_VA);
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
(8) : Rt<7:0>;
    r.address = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch32.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// AArch32.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// When the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALL(AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr)
attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_VMALLTLBIOp_VAA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = (32) : Rt<31:12> : ZerosTLBILevel_AnyZeros;
    r.vmid = vmid;
    r.attr = attr;(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

**Library pseudocode for aarch64/instrs/system/sysops/  
tlbi/AArch32.TLBI\_VMALLS12AArch32.TLBI\_VMALL**

```
// AArch32.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// AArch32.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALLS12(AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                                           Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    EL1};

    TLBIRecord r;
    r.op          = TLBIOp_VMALLS12TLBIOp_VMALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBIlevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

**Library pseudocode for aarch64/instrs/system/sysops/  
tlbi/AArch64.TLBI\_ALLAArch32.TLBI\_VMALLS12**

```
// AArch64.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// AArch32.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ALL(AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, regime, bits(16) vmid, Shareability shareability, attr))
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op = TLBIOp_ALLTLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel_Any;
    r.vmid = vmid;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

```

// AArch64.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// AArch64.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ASID(AArch64.TLBI_ALL(SecurityState security, Regime regime, bits(16) vmid,regime, Shareabi
    TLBIMemAttr attr, bits(64) Xt)
attr)
    assert PSTATE.EL IN {EL3, EL2,}; EL1};
    TLBIRecord r;
    r.op = TLBIOp_ASIDTLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = TLBILevel_Any;
    r.attr = attr;
    r.asid = Xt<63:48>; r.attr = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch64.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// AArch64.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_IPAS2(AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                                     Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
  assert PSTATE.EL IN {EL3, EL2};

  EL1;

  TLBIRecord r;
  r.op          = TLBIOp_IPAS2TLBIOp_ASID;
  r.from_aarch64 = TRUE;
  r.security     = security;
  r.regime       = regime;
  r.vmid         = vmid;
  r.level        = level;
  r.attr         = attr;
  r.address      = r.level = ZeroExtendTLBILevel_Any(Xt<39:0> : Zeros(12));

  case security of
    when SS_NonSecure
      r.ipaspace = PAS_NonSecure;
    when SS_Secure
      r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
  r.attr      = attr;
  r.asid      = Xt<63:48>;

  TLBI(r);
  if shareability != Shareability_NSH then Broadcast(shareability, r);
  return;

```

**Library pseudocode for aarch64/instrs/system/sysops/  
tlbi/AArch64.TLBI\_RIPAS2AArch64.TLBI\_IPAS2**

```
// AArch64.TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// AArch64.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RIPAS2(AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                                         Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp_RIPAS2TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;

    bits(2) tg      = Xt<47:46>;
    integer scale    = r.address = UIntZeroExtend(Xt<45:44>);
    integer num      = (Xt<39:0> : UIntZeros(Xt<43:39>));
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;
(12));

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/instrs/system/sysops/ tlbi/AArch64.TLBI\_RVA

```
// AArch64.TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// AArch64.TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVA(AArch64.TLBI_RIPAS2(
    SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_RVATLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

```

// AArch64.TLBI_RVAA()
// =====
// AArch64.TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVAA(AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_RVATLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);
;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

```

// AArch64.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// AArch64.TLBI_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VA(AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp\_VATLBIOp\_RVAA;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = Xt<63:48>;
r.address =
bits(2) tg = Xt<47:46>;
integer scale = ZeroExtendUInt(Xt<43:0> : (Xt<45:44>));
integer num = (Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

boolean valid;

(valid, r.tg, r.address, r.end_address) = TLBIRangeZerosUInt(12));(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
return;

```

```

// AArch64.TLBI_VAA()
// =====
// AArch64.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VAA(AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                                Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_VATLBIOp_VA;
r.from_aarch64 = TRUE;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = level;
r.attr         = attr;
r.asid        = Xt<63:48>;
r.address      = ZeroExtend(Xt<43:0> : Zeros(12));

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

```

```

// AArch64.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// AArch64.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// When the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALL(AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr)
attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_VMALLTLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.level = r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = (Xt<43:0> : ZerosTLBILevel_AnyZeroExtend;
    r.vmid = vmid;
    r.attr = attr;(12));

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

```

## Library pseudocode for aarch64/instrs/system/sysops/ tlbi/AArch64.TLBI\_VMALLS12AArch64.TLBI\_VMALL

```
// AArch64.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// AArch64.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALLS12(AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                                           Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    EL1};

    TLBIRecord r;
    r.op          = TLBIOp_VMALLS12TLBIOp_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBIlevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/ASID\_NONEAArch64.TLBI\_VMALLS12

```
constant bits(16) // AArch64.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete. ASID_NONE = AArch64.TLBI_VMALLS12( security, Regime regime, bits(16) vmid,
//                                           Shareability shareability, TLBIMemAttr attr)
// =====
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBIlevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then BroadcastZerosSecurityState();(shareability, r);
    return;
```

### Library pseudocode for aarch64/instrs/system/sysops/tlbi/BroadcastASID\_NONE

```
// Broadcast()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain. constant bits(16)

Broadcast(ASID_NONE =ShareabilityZeros shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED();
```

### Library pseudocode for aarch64/instrs/system/sysops/tlbi/DecodeTLBITGBroadcast

```
// DecodeTLBITG()
// =====
// Decode translation granule size in TLBI range instructions

TGx// Broadcast()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain. DecodeTLBITG(bits(2) tg)
    case tg of
        when '01' returnBroadcast( TGx_4KBShareability;
        when '10' returnshareability, TGx_16KBTLBIRecord;
        when '11' return TGx_64KB;r)
    IMPLEMENTATION_DEFINED;
```

### Library pseudocode for aarch64/instrs/system/sysops/tlbi/HasLargeAddressDecodeTLBITG

```
// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.
// DecodeTLBITG()
// =====
// Decode translation granule size in TLBI range instructions

booleanTGx HasLargeAddress(DecodeTLBITG(bits(2) tg)
    case tg of
        when '01' returnRegimeTGx_4KB regime)
    if !;
        when '10' returnHave52BitIPAAAndPASpaceExtTGx_16KB() then
            return FALSE;
    case regime of
        when;
        when '11' return Regime_EL3TGx_64KB
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL20
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();;
```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIHasLargeAddress

```
// TLBI()
// =====
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.
// =====
// Returns TRUE if the regime is configured for 52-bit addresses, FALSE otherwise.

boolean

TLBI(HasLargeAddress( regime)
    if !Have52BitIPAAAndPASpaceExt() then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL20
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            UnreachableTLBIRecordRegime r)
IMPLEMENTATION_DEFINED;();
```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBILevelTLBI

```
enumeration// TLBI()
// =====
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.
    TLBILevel_Any,
    TLBILevel_Last
};r)
IMPLEMENTATION_DEFINED;
```



```

// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

boolean enumeration TLBIMatch(TLBILevel {TLBIRecord tlbi, TLBILevel_Any, TLBRecord entry)
    boolean match;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL
            match = (tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime);
        when TLBIOp_DASID, TLBIOp_IASID
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (UseASID(entry.context) && entry.context.nG == '1' &&
                     tlbi.asid == entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                     entry.context.nG == '0') &&
                    tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_ALL
            relax_regime = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                            entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == entry.context.ss &&
                    (tlbi.regime == entry.context.regime || relax_regime));
        when TLBIOp_ASID
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (UseASID(entry.context) && entry.context.nG == '1' &&
                     tlbi.asid == entry.context.asid));
        when TLBIOp_IPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    tlbi.ipaspace == entry.context.ipaspace &&
                    tlbi.address<51:entry.blocksize> == entry.context.ia<51:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VAA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                     entry.context.nG == '0') &&
                    tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VMALL
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
        when TLBIOp_VMALLS12
            match = (tlbi.security == entry.context.ss &&

```

```

        tlbi.regime == entry.context.regime &&
        (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
    when TLBIOp_RIPAS2
        match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
            tlbi.security == entry.context.ss &&
            tlbi.regime == entry.context.regime &&
            (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
            tlbi.ipaspace == entry.context.ipaspace &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
            UInt(tlbi.address) <= UInt(entry.context.ia) &&
            UInt(tlbi.end_address) > UInt(entry.context.ia));
    when TLBIOp_RVAA
        match = (entry.context.includes_s1 &&
            tlbi.security == entry.context.ss &&
            tlbi.regime == entry.context.regime &&
            (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
            UInt(tlbi.address) <= UInt(entry.context.ia) &&
            UInt(tlbi.end_address) > UInt(entry.context.ia));
    when TLBIOp_RVA
        match = (entry.context.includes_s1 &&
            tlbi.security == entry.context.ss &&
            tlbi.regime == entry.context.regime &&
            (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
            (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                entry.context.nG == '0') &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
            UInt(tlbi.address) <= UInt(entry.context.ia) &&
            UInt(tlbi.end_address) > UInt(entry.context.ia));

    if tlbi.attr == TLBI_ExcludeXS && entry.context.xs == '1' then
        match = FALSE;

    return match; TLBIlevel_Last
};

```



```

enumeration// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMemAttr {TLBIMatch(
    TLBI_AllAttr,tlbi,
    entry)
    boolean match;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL
            match = (tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime);
        when TLBIOp_DASID, TLBIOp_IASID
            match = (entry.context.includes_s1 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                (UseASID(entry.context) && entry.context.nG == '1' &&
                    tlbi.asid == entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA
            match = (entry.context.includes_s1 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                    entry.context.nG == '0') &&
                tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                (tlbi.level == TLBIlevel_Any || !entry.walkstate.istable));
        when TLBIOp_ALL
            relax_regime = (tlbi.from_aarch64 &&
                tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == entry.context.ss &&
                (tlbi.regime == entry.context.regime || relax_regime));
        when TLBIOp_ASID
            match = (entry.context.includes_s1 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                (UseASID(entry.context) && entry.context.nG == '1' &&
                    tlbi.asid == entry.context.asid));
        when TLBIOp_IPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                tlbi.ipaspace == entry.context.ipaspace &&
                tlbi.address<51:entry.blocksize> == entry.context.ia<51:entry.blocksize> &&
                (tlbi.level == TLBIlevel_Any || !entry.walkstate.istable));
        when TLBIOp_VAA
            match = (entry.context.includes_s1 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                (tlbi.level == TLBIlevel_Any || !entry.walkstate.istable));
        when TLBIOp_VA
            match = (entry.context.includes_s1 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                    entry.context.nG == '0') &&
                tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                (tlbi.level == TLBIlevel_Any || !entry.walkstate.istable));
        when TLBIOp_VMALL
            match = (entry.context.includes_s1 &&
                tlbi.security == entry.context.ss &&
                tlbi.regime == entry.context.regime &&
                (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
    end case
end boolean

```

```

when TLBIOp_VMALLS12
    match = (tlbi.security == entry.context.ss &&
              tlbi.regime == entry.context.regime &&
              (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
when TLBIOp_RIPAS2
    match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime == entry.context.regime &&
              (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              tlbi.ipaspace == entry.context.ipaspace &&
              (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
              UInt(tlbi.address) <= UInt(entry.context.ia) &&
              UInt(tlbi.end_address) > UInt(entry.context.ia));
when TLBIOp_RVAA
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime == entry.context.regime &&
              (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
              UInt(tlbi.address) <= UInt(entry.context.ia) &&
              UInt(tlbi.end_address) > UInt(entry.context.ia));
when TLBIOp_RVA
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime == entry.context.regime &&
              (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
               entry.context.nG == '0') &&
              (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
              UInt(tlbi.address) <= UInt(entry.context.ia) &&
              UInt(tlbi.end_address) > UInt(entry.context.ia));

if tlbi.attr == TLBI_ExcludeXS TLBI_ExcludeXS
]; && entry.context.xs == '1' then
    match = FALSE;

return match;

```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIOpTLBIMemAttr

```

enumeration TLBIOp {TLBIMemAttr {
    TLBIOp_DALL, // AArch32 Data TLBI operations - deprecated TLBI_AllAttr,
    TLBIOp_DASID, TLBI_ExcludeXS
};
    TLBIOp_DVA,
    TLBIOp_IALL, // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
};

```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRangeTLBIOp

```
// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) enumeration TLBIRange(TLBIOp {Regime regime, bits(64) Xt)
    boolean valid = TRUE;
    bits(64) start = TLBIOp_DALL, // AArch32 Data TLBI operations - deprecated Zeros(64);
    bits(64) end = TLBIOp_DASID, Zeros(64);

    bits(2) tg = Xt<47:46>;
    integer scale = TLBIOp_DVA, UInt(Xt<45:44>);
    integer num = TLBIOp_IALL, // AArch32 Instruction TLBI operations - deprecated UInt(Xt<43:42>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start, end);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if TLBIOp_IASID, HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = TLBIOp_IVA, Replicate(Xt<36>, 11);
            else
                start<48:12> = Xt<36:0>;
                start<63:49> = TLBIOp_ALL, Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if TLBIOp_ASID, HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = TLBIOp_IPAS2, Replicate(Xt<36>, 11);
            else
                start<50:14> = Xt<36:0>;
                start<63:51> = TLBIOp_VAA, Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start<52:16> = Xt<36:0>;
            start<63:53> = TLBIOp_VA, Replicate(Xt<36>, 11);
        otherwise TLBIOp_VMALL,
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end = start + range<63:0>;

    if end<52> != start<52> then
        // overflow, saturate it
        end = TLBIOp_VMALLS12, Replicate(start<52>, 64-52) : TLBIOp_RIPAS2, TLBIOp_RVAA,
        Ones(52);

    return (valid, tg, start, end); TLBIOp_RVA,
};
```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRecordTLBIRange

```

type// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRecord is (TLBIRange(
    TLBIOpRegime    op,
    boolean          from_aarch64, // originated as an AArch64 operation regime, bits(64) Xt)
    boolean valid = TRUE;
    bits(64) start =
        SecurityStateZeros security, {64};
    bits(64) end =
        RegimeZeros regime,
    bits(16) vmid,
    bits(16) asid, {64};

    bits(2) tg = Xt<47:46>;
    integer scale =
        TLBILevelUInt level, {Xt<45:44>};
    integer num =
        TLBIMemAttrUInt attr, {Xt<43:39>};
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start, end);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if
                (regime) then
                    start<52:16> = Xt<36:0>;
                    start<63:53> = Replicate(Xt<36>, 11);
                else
                    start<48:12> = Xt<36:0>;
                    start<63:49> = Replicate(Xt<36>, 15);
            when '10' // 16KB
                tg_bits = 14;
                if HasLargeAddress(regime) then
                    start<52:16> = Xt<36:0>;
                    start<63:53> = Replicate(Xt<36>, 11);
                else
                    start<50:14> = Xt<36:0>;
                    start<63:51> = Replicate(Xt<36>, 13);
            when '11' // 64KB
                tg_bits = 16;
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            otherwise
                Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end = start + range<63:0>;

    if end<52> != start<52> then
        // overflow, saturate it
        end = Replicate(start<52>, 64-52) : OnesPASpaceHasLargeAddress ipaspace, // For opera
    bits(64) address, // input address, for range operations, start address
    bits(64) end_address, // for range operations, end address
    bits(2) tg, // for range operations, translation granule
)(52);

    return (valid, tg, start, end);

```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMIDTLBIRecord

```
// VMID[]
// =====
// Effective VMID.

bits(16) type VMID[]
    if TLBIRecord is ( EL2EnabledTLBIOp() ) then
        if !op,
            boolean from_aarch64, // originated as an AArch64 operation ELUsingAArch32SecurityState(secure)
            if regime,
                bits(16) vmid,
                bits(16) asid, Have16bitVMIDTLBILevel() && VTCR_EL2.VS == '1' then
                    return VTTBR_EL2.VMID;
                else
                    return level, ZeroExtendTLBIMemAttr(VTTBR_EL2.VMID<7:0>, 16);
            else
                return attr, ZeroExtendPASpace(VTTBR.VMID, 16);
        elsif HaveEL(EL2) && HaveSecureEL2Ext() then
            return Zeros(16);
        else
            return VMID_NONE; ipaspace, // For operations that take IPA as input address
            bits(64) address, // input address, for range operations, start address
            bits(64) end_address, // for range operations, end address
            bits(2) tg, // for range operations, translation granule
    )
```

## Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID\_NONEVMID

```
constant bits(16) // VMID[]
// =====
// Effective VMID.

bits(16) VMID_NONE = VMID[]
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if Have16bitVMID() && VTCR_EL2.VS == '1' then
                return VTTBR_EL2.VMID;
            else
                return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            return ZeroExtend(VTTBR.VMID, 16);
    elsif HaveEL(EL2) && HaveSecureEL2Ext() then
        return Zeros(16);
    else
        return VMID_NONE();
```

## Library pseudocode for aarch64/instrs/vectorsystem/arithmetic/sysops/binary/tlbi/uniform/logical/bsl-eor/vbitop/VBitOpVMID\_NONE

```
enumeration constant bits(16) VBitOp {VMID_NONE = VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

## Library pseudocode for aarch64/instrs/vector/arithmetic/unarybinary/cmpuniform/compareoplogical/CompareOpbsl-eor/vbitop/VBitOp

```
enumeration CompareOp {VBitOp {CompareOp_GT, VBitOp_VBIF, CompareOp_GE, VBitOp_VBIT, CompareOp_EQ, VBitOp_VBSL, CompareOp_LE, VBitOp_VEOR}; CompareOp_LT};
```

## Library pseudocode for aarch64/instrs/vector/logicalarithmetic/immediateopunary/ImmediateOpcmp/compareop/CompareOp

```
enumeration ImmediateOp {CompareOp {ImmediateOp_MOVI, CompareOp_GT, ImmediateOp_MVNI, CompareOp_GE, ImmediateOp_ORR, CompareOp_EQ, ImmediateOp_BIC}; CompareOp_LE, CompareOp_LT};
```

## Library pseudocode for aarch64/instrs/ vector/reducelogical/reduceopimmediateop/ReduceImmediateOp

```
// Reduce()
// =====

bits(esize) enumeration Reduce(ImmediateOp {ReduceOp op, bits(N) input, integer esize)
    boolean altfp = ImmediateOp_MOVI, HaveAltFP() && !ImmediateOp_MVNI, UsingAArch32() && FPCR.AH == '1';
    return ImmediateOp_ORR, Reduce(op, input, esize, altfp);

// Reduce()
// =====
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result; ImmediateOp_BIC};
```

## Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOpReduce

```

enumeration // Reduce()
// =====

bits(esize) ReduceOp {Reduce(ReduceOp_FMINNUM, op, bits(N) input, integer esize)
    boolean altfp = ReduceOp_FMAXNUM, () && !
                    ReduceOp_FMIN, () && FPCR.AH == '1';
    return ReduceOp_FMAX, (op, input, esize, altfp);

// Reduce()
// =====
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize)
    ReduceOp_FADD, Reduce( op, bits(N) input, integer esize, boolean altfp)

    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMINNUM
            result = FPMInNum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaXNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMIn(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMaX(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result;

```

## Library pseudocode for aarch64/translationinstrs/debugvector/AArch64.CheckBreakpointreduce/reduceop/ReduceOp

```
// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord enumeration AArch64.CheckBreakpoint(bits(64) vaddress, ReduceOp { AccType acctype_in, integer s
    assert !ReduceOp_FMINNUM, ELUsingAArch32(ReduceOp_FMAXNUM, S1TranslationRegime());
    assert (ReduceOp_FMIN, UsingAArch32() && size IN {2,4}) || size == 4; ReduceOp_FMAX,
    AccType acctype = acctype_in;

    match = FALSE;

    for i = 0 to ReduceOp_FADD, NumBreakpointsImplemented() - 1
        match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elseif match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault(); ReduceOp_ADD};
```

## Library pseudocode for aarch64/translation/ debug/AArch64.CheckDebugAArch64.CheckBreakpoint

```
// AArch64.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.
// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype)
    assert !

    FaultRecordELUsingAArch32 fault = ( NoFaultS1TranslationRegime();
    boolean generate_exception;

    boolean d_side = ({});
    assert (IsDataAccessUsingAArch32(acctype) || acctype == () && size IN {2,4}) || size == 4; AccType_DC
    boolean i_side = (acctype == acctype_in;

    match = FALSE;

    for i = 0 to AccType_IFETCHNumBreakpointsImplemented;
    if() - 1
        match_i = HaveNV2ExtAArch64.BreakpointMatch() && acctype == (i, vaddress, acctype, size);
        match = match || match_i;

    if match && AccType_NV2REGISTERHaltOnBreakpointOrWatchpoint then
        mask = '0';
        ss = () then
            reason = CurrentSecurityStateDebugHalt_Breakpoint();
            generate_exception =; AArch64.GenerateDebugExceptionsFromHalt((reason);
        elsif match then
            acctype = EL2AccType_IFETCH, ss, mask) && MDSCR_EL1.MDE == '1';
        else
            generate_exception =;
            iswrite = FALSE;
            return AArch64.GenerateDebugExceptionsAArch64.DebugFault() && MDSCR_EL1.MDE == '1';
        halt =(acctype, iswrite);
    else
        return HaltOnBreakpointOrWatchpointNoFault();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        elsif i_side then
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);

    return fault;();
```

**Library pseudocode for aarch64/translation/  
debug/AArch64.CheckWatchpointAArch64.CheckDebug**

```
// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.
// AArch64.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AArch64.CheckDebug(bits(64) vaddress, AccType acctype,
    boolean iswrite_in, integer size)
    assert !acctype, boolean iswrite, integer size) ELUsingAArch32FaultRecord(fault = S1TranslationRegime(
    boolean iswrite = iswrite_in;

    if acctype == AccType_DC then
        if !iswrite then
            return NoFault();
    elseif !boolean generate_exception;

    d_side = (acctype != IsDataAccessAccType_IFETCH(acctype) then
        return);
    if NoFaultHaveNV2Ext();

    match = FALSE;
    match_on_read = FALSE;
    ispriv =() && acctype == AArch64.AccessUsesELAccType_NV2REGISTER(acctype) !=then
        mask = '0';
        generate_exception = EL0AArch64.GenerateDebugExceptionsFrom;

    for i = 0 to( NumWatchpointsImplementedEL2() - 1
        if, AArch64.WatchpointMatchIsSecure(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
            if DBGWCR_EL1[i].LSC<0> == '1' then
                match_on_read = TRUE;

    if match &&(, mask) && MDSCR_EL1.MDE == '1';
    else
        generate_exception = IsAtomicRWAArch64.GenerateDebugExceptions(acctype) then
            iswrite = !match_on_read;

    if match &&(, MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint() then
        reason =();

    if generate_exception || halt then
        if d_side then
            fault = DebugHalt_WatchpointAArch64.CheckWatchpoint;
            EDWAR = vaddress;{vaddress, acctype, iswrite, size};
        else
            fault =
                HaltAArch64.CheckBreakpoint(reason);
    elseif match then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault();{vaddress, acctype, size};

    return fault;
```

## Library pseudocode for aarch64/

translation/vmsa\_addrcalcdebug/AArch64.BlockBaseAArch64.CheckWatchpoint

```
// AArch64.BlockBase()
// =====
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block
// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

bits(52) FaultRecord AArch64.BlockBase(bits(64) descriptor, bit ds, AArch64.CheckWatchpoint(bits(64) vaddress,
    bits(52) blockbase = acctype,
    boolean iswrite_in, integer size)
assert ! ZerosELUsingAArch32();

if tgx == ( TGx_4KBS1TranslationRegime && level == 2 then
    blockbase<47:21> = descriptor<47:21>;
elseif tgx == ();
    boolean iswrite = iswrite_in;

if acctype IN { TGx_4KBAccType_TTW && level == 1 then
    blockbase<47:30> = descriptor<47:30>;
elseif tgx ==, TGx_4KBAccType_IC && level == 0 then
    blockbase<47:39> = descriptor<47:39>;
elseif tgx ==, TGx_16KBAccType_AT && level == 2 then
    blockbase<47:25> = descriptor<47:25>;
elseif tgx ==, TGx_16KBAccType_ATPAM && level == 1 then
    blockbase<47:36> = descriptor<47:36>;
elseif tgx ==} then
    return TGx_64KBNoFault && level == 2 then
        blockbase<47:29> = descriptor<47:29>;
elseif tgx ==();
if acctype == TGx_64KBAccType_DC && level == 1 then
    blockbase<47:42> = descriptor<47:42>;
else then
    if !iswrite then
        return
        UnreachableNoFault();

if match = FALSE;
match_on_read = FALSE;
ispriv = Have52BitPAExtAArch64.AccessUsesEL() && tgx == (acctype) != ;

for i = 0 to NumWatchpointsImplemented() - 1
    if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
        match = TRUE;
        if DBGWCR_EL1[i].LSC<0> == '1' then
            match_on_read = TRUE;

if match && acctype == AccType_ATOMICRW then
    iswrite = !match_on_read;

if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Watchpoint;
    EDWAR = vaddress;
    Halt(reason);
elseif match then
    return AArch64.DebugFault(acctype, iswrite);
else
    return NoFaultTGx_64KBEL0 then
        blockbase<51:48> = descriptor<15:12>;
    elseif ds == '1' then
        blockbase<51:48> = descriptor<9:8>;descriptor<49:48>;

return blockbase();
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.IASizeAArch64.BlockBase

```
// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address
// AArch64.BlockBase()
// =====
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block

integer bits(52) AArch64.IASize(bits(6) txsz)
return 64 - AArch64.BlockBase(bits(64) descriptor, bit ds, txsz, integer level)
bits(52) blockbase = Zeros();

if txsz == TGx_4KB && level == 2 then
    blockbase<47:21> = descriptor<47:21>;
elsif txsz == TGx_4KB && level == 1 then
    blockbase<47:30> = descriptor<47:30>;
elsif txsz == TGx_4KB && level == 0 then
    blockbase<47:39> = descriptor<47:39>;
elsif txsz == TGx_16KB && level == 2 then
    blockbase<47:25> = descriptor<47:25>;
elsif txsz == TGx_16KB && level == 1 then
    blockbase<47:36> = descriptor<47:36>;
elsif txsz == TGx_64KB && level == 2 then
    blockbase<47:29> = descriptor<47:29>;
elsif txsz == TGx_64KB && level == 1 then
    blockbase<47:42> = descriptor<47:42>;
else
    Unreachable();

if Have52BitPAExt() && txsz == TGx_64KB then
    blockbase<51:48> = descriptor<15:12>;
elsif ds == '1' then
    blockbase<51:48> = descriptor<9:8>; descriptor<49:48>;

return blockbase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.NextTableBaseAArch64.IASize

```
// AArch64.NextTableBase()
// =====
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors
// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address

bits(52) integer AArch64.NextTableBase(bits(64) descriptor, bit ds, AArch64.IASize(bits(6) txsz)
return 64 - TGxUInt txsz)
bits(52) tablebase = Zeros();

case txsz of
    when TGx_4KB tablebase<47:12> = descriptor<47:12>;
    when TGx_16KB tablebase<47:14> = descriptor<47:14>;
    when TGx_64KB tablebase<47:16> = descriptor<47:16>;

if Have52BitPAExt() && txsz == TGx_64KB then
    tablebase<51:48> = descriptor<15:12>;
elsif ds == '1' then
    tablebase<51:48> = descriptor<9:8>; descriptor<49:48>;

return tablebase; txsz;
```

**Library pseudocode for aarch64/translation/  
vmsa\_addrcalc/AArch64.PageBaseAArch64.NextTableBase**

```
// AArch64.PageBase()
// =====
// Extract the address embedded in a page descriptor pointing to the base of
// a memory page
// AArch64.NextTableBase()
// =====
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(52) AArch64.PageBase(bits(64) descriptor, bit ds, AArch64.NextTableBase(bits(64) descriptor, bit ds,
bits(52) pagebase = bits(52) tablebase = Zeros();

    case tgx of
        when TGx_4KB pagebase<47:12> = descriptor<47:12>;
tablebase<47:12> = descriptor<47:12>;
        when TGx_16KB pagebase<47:14> = descriptor<47:14>;
tablebase<47:14> = descriptor<47:14>;
        when TGx_64KB pagebase<47:16> = descriptor<47:16>;
tablebase<47:16> = descriptor<47:16>;

    if Have52BitPAExt() && tgx == TGx_64KB then
        pagebase<51:48> = descriptor<15:12>;
tablebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        pagebase<51:48> = descriptor<9:8>:descriptor<49:48>;
tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return pagebase; return tablebase;
```

## Library pseudocode for aarch64/translation/ vmsa\_addrcalc/AArch64.PhysicalAddressSizeAArch64.PageBase

```
// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address
// AArch64.PageBase()
// =====
// Extract the address embedded in a page descriptor pointing to the base of
// a memory page

integer bits(52) AArch64.PhysicalAddressSize(bits(3) encoded_ps, AArch64.PageBase(bits(64) descriptor, bit
integer ps;
integer max_ps;

case encoded_ps of
    when '000' ps = 32;
    when '001' ps = 36;
    when '010' ps = 40;
    when '011' ps = 42;
    when '100' ps = 44;
    when '101' ps = 48;
    when '110' ps = 52;
    otherwise
        ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if tgx != bits(52) pagebase = TGx_64KBZeros &&!();

case tgx of
    when Have52BitIPAAAndPASpaceExtTGx_4KB() then
        max_ps = pagebase<47:12> = descriptor<47:12>;
    when MinTGx_16KB(48, pagebase<47:14> = descriptor<47:14>;
    when AArch64.PAMaxTGx_64KB();
else
    max_ps = pagebase<47:16> = descriptor<47:16>;

if AArch64.PAMaxHave52BitPAExt();

return() && tgx == MinTGx_64KB(ps, max_ps); then
    pagebase<51:48> = descriptor<15:12>;
elseif ds == '1' then
    pagebase<51:48> = descriptor<9:8>:descriptor<49:48>;

return pagebase;
```

**Library pseudocode for aarch64/translation/  
vmsa\_addrcalc/AArch64.S1StartLevelAArch64.PhysicalAddressSize**

```
// AArch64.S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translation
// table walk
// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

integer AArch64.S1StartLevel(AArch64.PhysicalAddressSize(bits(3) encoded_ps, S1TTWParamsTGx walkparams)
// Input Address size
iasize = tgsz;
integer ps;
integer max_ps;

case encoded_ps of
    when '000' ps = 32;
    when '001' ps = 36;
    when '010' ps = 40;
    when '011' ps = 42;
    when '100' ps = 44;
    when '101' ps = 48;
    when '110' ps = 52;
    otherwise
        ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if tgsz != AArch64.IASizeTGx_64KB(walkparams.tgsz);
granulebits = && ! TGxGranuleBitsHave52BitIPAAAndPASpaceExt(walkparams.tgsz);
stride = granulebits - 3;

return() then
    max_ps = (48, AArch64.PAMax());
else
    max_ps = AArch64.PAMax();

return MinFINAL_LEVELMin - (((iasize-1) - granulebits) DIV stride);(ps, max_ps);
```

**Library pseudocode for aarch64/translation/  
vmsa\_addrcalc/AArch64.S2SLTTEnterAddressAArch64.S1StartLevel**

```
// AArch64.S2SLTTEnterAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level
// AArch64.S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translation
// table walk

FullAddress integer AArch64.S2SLTTEnterAddress(AArch64.S1StartLevel(S2TTWParams S1TTWParams walkparams, bit
// Input Address size
iasize =
FullAddress tablebase)
startlevel = AArch64.S2StartLevel(walkparams);
iasize = AArch64.IASize(walkparams.txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
levels =
return FINAL_LEVEL - startlevel;

bits(52) index;
lsb = levels*stride + granulebits;
msb = iasize - 1;
index = ZeroExtend(ipa<msb:lsb>:Zeros(3));

FullAddress descaddress;
descaddress.address = tablebase.address OR index;
descaddress.paspace = tablebase.paspace;

return descaddress; -- (((iasize-1) -- granulebits) DIV stride);
```

## Library pseudocode for aarch64/translation/ vmsa\_addrcalc/AArch64.S2StartLevelAArch64.S2SLTTEntryAddress

```
// AArch64.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk
// AArch64.S2SLTTEntryAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

integer FullAddress AArch64.S2StartLevel(AArch64.S2SLTTEntryAddress(S2TTWParams walkparams)
    case walkparams.tgx of
        when walkparams, bits(52) ipa, TGx_4KBFullAddress
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
            when tablebase)
        startlevel = TGx_16KBAArch64.S2StartLevel
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
            when(walkparams);
        iasize = (walkparams.txsz);
        granulebits = TGxGranuleBits(walkparams.tgx);
        stride = granulebits - 3;
        levels = FINAL_LEVEL - startlevel;

        bits(52) index;
        lsb = levels*stride + granulebits;
        msb = iasize - 1;
        index = ZeroExtend(ipa<msb:lsb>:Zeros(3));

        FullAddressTGx_64KBAArch64.IASize
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;descaddress;
        descaddress.address = tablebase.address OR index;
        descaddress.paspace = tablebase.paspace;

        return descaddress;
```

**Library pseudocode for aarch64/translation/  
vmsa\_addrcalc/AArch64.TTBaseAddressAArch64.S2StartLevel**

```
// AArch64.TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table
// AArch64.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

bits(52) integer AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz, bits(3) ps,
                                         bit ds, AArch64.S2StartLevel( TGxS2TTWParams tgx, integer startlevel))
    bits(52) tablebase = walkparams;
    case walkparams.tgx of
        when ZerosTGx_4KB();

    // Input Address size
    iasize = case walkparams.sl2:walkparams.sl0 of
        when '000' return 2;
        when '001' return 1;
        when '010' return 0;
        when '011' return 3;
        when '100' return -1;
    when AArch64.IASizeTGx_16KB(txsz);
    granulebits = case walkparams.sl0 of
        when '00' return 3;
        when '01' return 2;
        when '10' return 1;
        when '11' return 0;
    when TGxGranuleBits(tgx);
    stride = granulebits - 3;
    levels = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + 3;

    if (Have52BitPAExt() && tgx == TGx_64KB && ps == '110') || (ds == '1') then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttb<5:2>;ttb<47:6>;
    else
        tablebase<47:1> = ttb<47:1>;

    tablebase = Align(tablebase, 1 << tsize);
    return tablebase; case walkparams.sl0 of
        when '00' return 3;
        when '01' return 2;
        when '10' return 1;
```

**Library pseudocode for aarch64/translation/  
vmsa\_addrcalc/AArch64.TTEntryAddressAArch64.TTBaseAddress**

```
// AArch64.TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base
// AArch64.TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table

FullAddressbits(52) AArch64.TTEntryAddress(integer level, AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz,
bit ds, TGx tgx, bits(6) txsz,
bits(64) ia, tgx, integer startlevel)
bits(52) tablebase = FullAddressZeros tablebase)
());

// Input Address size
iasize = AArch64.IASize(txsz);
granulebits = TGxGranuleBits(tgx);
stride = granulebits - 3;
levels = FINAL_LEVEL - level;
- startlevel;

bits(52) index;
lsb = levels*stride + granulebits;
msb = // Base address is aligned to size of the initial translation table in bytes
tsize = (iasize - (levels*stride + granulebits)) + 3;

if ( MinHave52BitPAExt(iasize - 1, (lsb + stride) - 1);
index =() && tgx == ZeroExtendTGx_64KB(ia<msb:lsb>:&& ps == '110') || (ds == '1') then
tsize =ZerosMax(3));(tsize, 6);
tablebase<51:6> = ttb<5:2>:ttb<47:6>;
else
tablebase<47:1> = ttb<47:1>;

tablebase =

FullAddressAlign descaddress;
descaddress.address = tablebase.address OR index;
descaddress.paspace = tablebase.paspace;

return descaddress;(tablebase, 1 << tsize);
return tablebase;
```

## Library pseudocode for aarch64/

translation/vmsa\_faults/vmsa\_addrcalc/AArch64.AddrTopAArch64.TTEntryAddress

```
// AArch64.AddrTop()
// =====
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.
// AArch64.TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base

integer FullAddress AArch64.AddrTop(bit tbid, AArch64.TTEntryAddress(integer level, AccType TGx acctype, bit
    if tbid == '1' && acctype == tgx, bits(6) txsz,
        bits(64) ia, tablebase)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride = granulebits - 3;
    levels = FINAL_LEVEL - level;

    bits(52) index;
    lsb = levels*stride + granulebits;
    msb = Min(iasize - 1, (lsb + stride) - 1);
    index = ZeroExtend(ia<msb:lsb>:Zeros(3));

    FullAddress AccType_IFETCH FullAddress then
        return 63;
descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    if tbi == '1' then
        return 55;
    else
        return 63; return descaddress;
```

## Library pseudocode for aarch64/translation/

vmsa\_faults/AArch64.ContiguousBitFaultsAArch64.AddrTop

```
// AArch64.ContiguousBitFaults()
// =====
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault
// AArch64.AddrTop()
// =====
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

boolean integer AArch64.ContiguousBitFaults(bits(6) txsz, AArch64.AddrTop(bit tbid, TGx AccType tgx, integer
    // Input Address size
    iasize = acctype, bit tbi)
    if tbid == '1' && acctype == AArch64.IASize AccType_IFETCH(txsz);
    // Translation size
    tsize = TranslationSize(tgx, level) + ContiguousSize(tgx, level);
then
    return 63;

    fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit";

    return tsize > iasize && fault; if tbi == '1' then
        return 55;
    else
        return 63;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.DebugFaultAArch64.ContiguousBitFaults**

```
// AArch64.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint
// AArch64.ContiguousBitFaults()
// =====
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

FaultRecord boolean AArch64.DebugFault(AArch64.ContiguousBitFaults(bits(6) txsz, AccType TGx acctype, boolean iswrite)
// Input Address size
iasize =
    FaultRecord AArch64.IASize fault;

    fault.statuscode = (txsz);
// Translation size
tsize = (tgx, level) + ContiguousSizeFault_DebugTranslationSize;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
(tgx, level);

    return fault; // fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous
// return tsize > iasize && fault;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.ExclusiveFaultAArch64.DebugFault**

```
// AArch64.ExclusiveFault()
// =====
// AArch64.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch64.ExclusiveFault(AArch64.DebugFault(AccType acctype, boolean iswrite,
    boolean secondstage, boolean s2fslwalk) acctype, boolean iswrite)
    FaultRecord fault;

    fault.statuscode = Fault_ExclusiveFault_Debug;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = secondstage;
    fault.s2fslwalk = s2fslwalk;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    return fault;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.IPAIsOutOfRangeAArch64.ExclusiveFault**

```
// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO
// AArch64.ExclusiveFault()
// =====

boolean FaultRecord AArch64.IPAIsOutOfRange(bits(52) ipa, AArch64.ExclusiveFault( S2TTWParams AccType walkpa
    //Input Address size
    iasize = acctype, boolean iswrite,
    boolean secondstage, boolean s2fslwalk) AArch64.IASizeFaultRecord(walkpa
fault;

    if iasize < 52 then
        return ! fault.statuscode = IsZeroFault_Exclusive(ipa<51:iasize>);
    else
        return FALSE;;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = secondstage;
    fault.s2fslwalk = s2fslwalk;

    return fault;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.OAOutOfRangeAArch64.IPAIsOutOfRange**

```
// AArch64.OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number of bits
// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

boolean AArch64.OAOutOfRange(AArch64.IPAIsOutOfRange(bits(52) ipa, ITWState S2TTWParams walkstate, bits(3)
    //Input Address size
    iasize = TGxAArch64.IASize tgx, bits(64) ia)
    // Output Address size
    oasize = AArch64.PhysicalAddressSize(ps, tgx);

    if oasize < 52 then
        if walkstate.istable then
            baseaddress = walkstate.baseaddress.address;
            return !IsZero(baseaddress<51:oasize>);
        else
            // Output address
            oa = Stage0A(ia, tgx, walkstate);
            return !(walkparams.txsz);

    if iasize < 52 then
        return !IsZero(oa.address<51:oasize>);
    (ipa<51:iasize>);
    else
        return FALSE;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.S1HasAlignmentFaultAArch64.OAOutOfRange**

```
// AArch64.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory
// AArch64.OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.S1HasAlignmentFault(AArch64.OAOutOfRange(AccType TTWState acctype, boolean aligned,
bit ntlsmid, walkstate, bits(3) ps, MemoryAttributes TGx memattrs)
    if acctype == tgx, bits(64) ia)
// Output Address size
oasize = AccType_IFETCHAArch64.PhysicalAddressSize || memattrs.memtype !=(ps, tgx);

    if oasize < 52 then
        if walkstate.istable then
            baseaddress = walkstate.baseaddress.address;
            return ! MemType_DeviceIsZero then
                return FALSE;

    if acctype == (baseaddress<51:oasize>);
    else
        // Output address
        oa = AccType_A32LSMDStage0A && ntlsmid == '0' && memattrs.device !=(ia, tgx, walkstate);
        return ! DeviceType_GREIsZero then
            return TRUE;

    return !aligned || acctype == AccType_DCZVA; (oa.address<51:oasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.S1HasPermissionsFaultAArch64.S1HasAlignmentFault

```

// AArch64.S1HasPermissionsFault()
// =====
// Returns whether stage 1 access violates permissions of target memory
// AArch64.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasPermissionsFault(AArch64.S1HasAlignmentFault(Regime regime, SecurityState ss, ITWState
                                     S1TTWParams walkparams, boolean ispriv, AccType acctype,
                                     boolean iswrite)

    bit r;
    bit w;
    bit x;
    permissions = walkstate.permissions;

    if acctype, boolean aligned,
        bit ntlsmd, HasUnprivilegedMemoryAttributes(regime) then
        bit pr;
        bit pw;
        bit ur;
        bit uw;
        // Apply leaf permissions
        case permissions.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr, '0', '0', '0'); // Read-only, privileged access

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);

        pan_access = !(acctype IN {memattrs})
        if acctype == AccType_DC, AccType_IFETCH, || memattrs.memtype != AccType_AT, AccType_NV2REGISTER});
        if HavePANExt() && pan_access && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
            bit pan;
            if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
                CurrentSecurityState() == SS_Secure &&
                walkstate.baseaddress.paspace == PAS_NonSecure &&
                walkparams.sif == '1') then
                ux = '0';

            pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

            (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
        else
            // Apply leaf permissions
            case permissions.ap<2> of
                when '0' (r,w) = ('1','1'); // No effect
                when '1' (r,w) = ('1','0'); // Read-only

            // Apply hierarchical permissions
            case permissions.ap_table<1> of
                when '0' (r,w) = ( r , w ); // No effect
                when '1' (r,w) = ( r , '0'); // Read-only

            x = NOT(permissions.xn OR permissions.xn_table);

            // Prevent execution from writable locations if WXN is set
            x = x AND NOT(walkparams.wxn AND w);

```

```

// Prevent execution from Non-secure space by PE in secure state if SIF is set
if ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
    x = x AND NOT(walkparams.sif);

if acctype == AccType\_IFETCH then
    if (ConstrainUnpredictable\(Unpredictable\_INSTRDEVICE\) == Constraint\_FAULT &&
        walkstate.memattrs.memtype == MemType\_Device) then
        return TRUE;
then
    return FALSE;

    return x == '0';
elseif acctype == if acctype == AccType\_DCAccType\_A32LSMD then
    if iswrite then
        return w == '0';
    else
        // DC from privileged context which do no write cannot Permission fault
        return !ispriv && (r == '0' ||
            (& ntlsmid == '0' && memattrs.device != IsCMOWControlledInstructionDeviceType\_GRE() && wa
        elseif acctype == then
            return TRUE;

return !aligned || acctype == AccType\_ICAccType\_DCZVA then
    // IC instructions do not write
    assert !iswrite;
    impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

    // IC from privileged context cannot Permission fault
    return !ispriv && ((r == '0' && impdef_ic_fault) ||
        (IsCMOWControlledInstruction() && walkparams.crow == '1' && w == '0'));
elseif iswrite then
    return w == '0';
else
    return r == '0';

```

Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.S1InvalidTxSZAArch64.S1HasPermissionsFault

```

// AArch64.S1InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ
// AArch64.S1HasPermissionsFault()
// =====
// Returns whether stage 1 access violates permissions of target memory

boolean AArch64.S1InvalidTxSZ(AArch64.S1HasPermissionsFault(Regime regime, SecurityState ss, TTWState wa
                                S1TTWParams walkparams)
    mintxsz = walkparams, boolean ispriv, AArch64.S1MinTxSZAccType(walkparams.ds, walkparams.tgx);
    maxtxsz = acctype,
                                boolean iswrite)

    bit r;
    bit w;
    bit x;
    permissions = walkstate.permissions;

    if AArch64.MaxTxSZHasUnprivileged(walkparams.tgx);
(regime) then
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    // Apply leaf permissions
    case permissions.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

    return // Apply hierarchical permissions
    case permissions.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr, '0', '0', '0'); // Read-only, privileged access

    // Locations writable by unprivileged cannot be executed by privileged
    px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
    ux = NOT(permissions.uxn OR permissions.uxn_table);

    pan_access = !(acctype IN { UIntAccType_DC(walkparams.txsz) < mintxsz || , AccType_AT, AccType_I
    if HavePANExt() && pan_access && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
        bit pan;
        if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
            CurrentSecurityState() == SS_Secure &&
            walkstate.baseaddress.paspace == PAS_NonSecure &&
            walkparams.sif == '1') then
            ux = '0';

        pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
else
    // Apply leaf permissions
    case permissions.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only

    // Apply hierarchical permissions
    case permissions.ap_table<1> of
        when '0' (r,w) = ( r , w ); // No effect
        when '1' (r,w) = ( r , '0'); // Read-only

    x = NOT(permissions.xn OR permissions.xn_table);

    // Prevent execution from writable locations if WXN is set

```

```

x = x AND NOT(walkparams.wxn AND w);
// Prevent execution from Non-secure space by PE in secure state if SIF is set
if ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
    x = x AND NOT(walkparams.sif);

if acctype == AccType_IFETCH then
    if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&
        walkstate.memattrs.memtype == MemType_Device) then
        return TRUE;

    return x == '0';
elseif acctype == AccType_DC then
    if iswrite then
        return w == '0';
    else
        // DC from privileged context which do no write cannot permission fault
        return !ispriv && (r == '0' ||
            (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elseif acctype == AccType_IC then
    // IC instructions do not write
    assert !iswrite;
    impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

    // IC from privileged context cannot permission fault
    return !ispriv && ((r == '0' && impdef_ic_fault) ||
        (IsCMOWControlledInstruction() && walkparams.txsz > maxtxsz;()) && walkparams.cmow == '1');
elseif iswrite then
    return w == '0';
else
    return r == '0';

```

## Library pseudocode for aarch64/translation/

### vmsa\_faults/AArch64.S2HasAlignmentFaultAArch64.S1InvalidTxSZ

```

// AArch64.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory
// AArch64.S1InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S2HasAlignmentFault(AArch64.S1InvalidTxSZ(AArch64.S1TTWParams acctype, boolean aligned, walkparams)
    mintxsz = MemoryAttributesAArch64.S1MinTxSZ(memattrs)
    if acctype == (walkparams.ds, walkparams.tgx);
    maxtxsz = AccType_IFETCHAArch64.MaxTxSZ || memattrs.memtype != (walkparams.tgx);

    return MemType_DeviceUInt then
        return FALSE;

    return !aligned || acctype == (walkparams.txsz) < mintxsz || AccType_DCZVAUInt; (walkparams.txsz) > maxtxsz

```

## Library pseudocode for aarch64/translation/

### vmsa\_faults/AArch64.S2HasPermissionsFaultAArch64.S2HasAlignmentFault

```
// AArch64.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory
// AArch64.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasPermissionsFault(boolean s2fslwalk, AArch64.S2HasAlignmentFault( TTWState walkstate,
S2TTWParams walkparams, boolean ispriv, AccType acctype,
boolean iswrite)

    permissions = walkstate.permissions;
    memtype = walkstate.memattrs.memtype;

    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;

    bit px;
    bit ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if ispriv then px else ux;

    if s2fslwalk && walkparams.ptw == '1' && memtype == acctype, boolean aligned, MemType_DeviceMemoryAttr
        return TRUE;
    elsif acctype == memattrs)
    if acctype == AccType_IFETCH then
        constraint = || memattrs.memtype != ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        return x == '0';
    elsif acctype == return FALSE;

    return !aligned || acctype == AccType_DC AccType_DCZVA then
        // AArch32 DC maintenance instructions operating by VA cannot fault.
        if iswrite then
            return !ELUsingAArch32(EL1) && w == '0';
        else
            return ((!ispriv && !ELUsingAArch32(EL1) && r == '0') ||
                (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif acctype == AccType_IC then
        // IC instructions do not write
        assert !iswrite;
        impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

        return ((!ispriv && !ELUsingAArch32(EL1) && r == '0' && impdef_ic_fault) ||
            (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif iswrite then
        return w == '0';
    else
        return r == '0';;
```

Library pseudocode for aarch64/translation/  
vmsa\_faults/**AArch64.S2InconsistentSLAArch64.S2HasPermissionsFault**

```

// AArch64.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields
// AArch64.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch64.S2InconsistentSL(AArch64.S2HasPermissionsFault(boolean s2fslwalk, TTWState walkstate, Secu
                                S2TTWParams walkparams)
    startlevel =walkparams, boolean ispriv, AArch64.S2StartLevelAccType(walkparams);
    levels     =acctype,
                                boolean iswrite)
    permissions = walkstate.permissions;
    memtype     = walkstate.memattrs.memtype;

    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;

    bit px;
    bit ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if ispriv then px else ux;

    if s2fslwalk && walkparams.ptw == '1' && memtype == FINAL_LEVELMemType_Device - startlevel;
    granulebits =then
        return TRUE;
    elsif acctype == TGxGranuleBitsAccType_IFETCH(walkparams.tgx);
    stride       = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize       =then
        constraint = (Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
            return x == '0';
    elsif acctype == AccType_DC then
        // AArch32 DC maintenance instructions operating by VA cannot fault.
        if iswrite then
            return !ELUsingAArch32(EL1) && w == '0';
        else
            return ((!ispriv && !ELUsingAArch32(EL1) && r == '0') ||
                (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif acctype == AccType_IC then
        // IC instructions do not write
        assert !iswrite;
        impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

        return ((!ispriv && !ELUsingAArch32(EL1) && r == '0' && impdef ic_fault) ||
            (IsCMOWControlledInstructionAArch64.IASizeConstrainUnpredictable(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;() && walkparams.cmow == '1' && w == '0');
    elsif iswrite then
        return w == '0';
    else
        return r == '0';

```

## Library pseudocode for aarch64/translation/

### vmsa\_faults/AArch64.S2InvalidSLAArch64.S2InconsistentSL

```
// AArch64.S2InvalidSL()
// =====
// Detect invalid configuration of SL field
// AArch64.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InvalidSL(AArch64.S2InconsistentSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when startlevel = TGx_4KBAArch64.S2StartLevel
            case walkparams.sl2:walkparams.sl0 of
                when '1x1' return TRUE;
                when '11x' return TRUE;
                when '010' return(walkparams);
            levels = AArch64.PAMaxFINAL_LEVEL() < 44;
                when '011' return !startlevel;
            granulebits = HaveSmallTranslationTableExtTGxGranuleBits();
                otherwise return FALSE;
            when(walkparams.tgx);
            stride = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits // Bits directly mapped to output address
        + 1); // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize = TGx_16KBAArch64.IASize
        case walkparams.sl0 of
            when '11' return walkparams.ds == '0';
            when '10' return AArch64.PAMax() < 42;
            otherwise return FALSE;
        when TGx_64KB
            case walkparams.sl0 of
                when '11' return TRUE;
                when '10' return AArch64.PAMax() < 44;
                otherwise return FALSE; (walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.S2InvalidTxSZAArch64.S2InvalidSL**

```
// AArch64.S2InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ
// AArch64.S2InvalidSL()
// =====
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidTxSZ(AArch64.S2InvalidSL(S2TTWParams walkparams, boolean slaarch64)
    mintxs = walkparams)
    case walkparams.tgx of
        when AArch64.S2MinTxSZTgx_4KB(walkparams.ds, walkparams.tgx, slaarch64);
        maxtxsz = case walkparams.sl2:walkparams.sl0 of
            when '1x1' return TRUE;
            when '11x' return TRUE;
            when '010' return AArch64.MaxTxSZAArch64.PAMax(walkparams.tgx);

    return() < 44;
        when '011' return ! UIntHaveSmallTranslationTableExt(walkparams.txsz) < mintxs || ();
        otherwise return FALSE;
    when
        case walkparams.ds:walkparams.sl0 of
            when '011' return TRUE;
            when '010' return AArch64.PAMax() < 42;
            otherwise return FALSE;
        when Tgx_64KB
            case walkparams.sl0 of
                when '11' return TRUE;
                when '10' return AArch64.PAMaxUIntTgx_16KB(walkparams.txsz) > maxtxsz; () < 44;
                otherwise return FALSE;
```

**Library pseudocode for aarch64/translation/  
vmsa\_faults/AArch64.VAIsOutOfRangeAArch64.S2InvalidTxSZ**

```
// AArch64.VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted value
// AArch64.S2InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.VAIsOutOfRange(bits(64) va, AArch64.S2InvalidTxSZ( AccTypeS2TTWParams acctype, walkparams,
    mintxs = RegimeAArch64.S2MinTxSZ regime, (walkparams.ds, walkparams.tgx, slaarch64);
    maxtxsz = S1TTWParamsAArch64.MaxTxSZ walkparams)
    addrtop = (walkparams.tgx);

    return AArch64.AddrTopUInt(walkparams.tbid, acctype, walkparams.tbi);
// Input Address size
iasize = (walkparams.txsz) < mintxs || AArch64.IASizeUInt(walkparams.txsz);

if HasUnprivileged(regime) then
    if AArch64.GetVARange(va) == VARange_LOWER then
        return !IsZero(va<addrtop:iasize>);
    else
        return !IsOnes(va<addrtop:iasize>);
else
    return !IsZero(va<addrtop:iasize>); (walkparams.txsz) > maxtxsz;
```

## Library pseudocode for aarch64/

translation/vmsa\_memattrvmsa\_faults/AArch64.IsS2ResultTaggedAArch64.VAIsOutOfRange

```
// AArch64.IsS2ResultTagged()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory
// AArch64.VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.IsS2ResultTagged(AArch64.VAIsOutOfRange(bits(64) va, MemoryAttributesAccType s2_memattrs,
    return (
        s1_tagged &&
        (s2_memattrs.memtype == acctype, MemType_NormalRegime) &&
        (s2_memattrs.inner.attrs == regime, MemAttr_WBS1TTWParams) &&
        (s2_memattrs.inner.hints == walkparams)
        addrtop = MemHint_RWAAArch64.AddrTop) &&
        (!s2_memattrs.inner.transient) &&
        (s2_memattrs.outer.attrs == (walkparams.tbid, acctype, walkparams.tbi));
// Input Address size
iasize = MemAttr_WBAArch64.IASize) &&
        (s2_memattrs.outer.hints == (walkparams.txsz);

    if (regime) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            return !IsZero(va<addrtop:iasize>);
        else
            return !IsOnes(va<addrtop:iasize>);
    else
        return !IsZeroMemHint_RWAHasUnprivileged) &&
        (!s2_memattrs.outer.transient)
    ); (va<addrtop:iasize>);
```

Library pseudocode for aarch64/translation/  
vmsa\_memattr/AArch64.S2ApplyFWBMemAttrsAArch64.IsS2ResultTagged

```

// AArch64.S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.
// AArch64.IsS2ResultTagged()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

MemoryAttributes boolean AArch64.S2ApplyFWBMemAttrs(AArch64.IsS2ResultTagged(MemoryAttributes s1_memattrs,
bits(4) s2_attr, bits(2) s2_sh)s2_memattrs, boolean s1_tagged)

    return (
        s1_tagged &&
        (s2_memattrs.memtype ==
            MemoryAttributes memattrs;

        if s2_attr<2> == '0' then // S2 Device, S1 any
            s2_device = DecodeDevice(s2_attr<1:0>);
            memattrs.memtype = MemType_Device;
            if s1_memattrs.memtype == MemType_Device then
                memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
            else
                memattrs.device = s2_device;

        elsif s2_attr<1:0> == '11' then // S2 attr = S1 attr
            memattrs = s1_memattrs;

        elsif s2_attr<1:0> == '10' then // Force writeback
            memattrs.memtype = MemType_Normal;
            memattrs.inner.attrs =) &&
            (s2_memattrs.inner.attrs == MemAttr_WB;
            memattrs.outer.attrs =) &&
            (s2_memattrs.inner.hints == MemAttr_WB;

            if (s1_memattrs.memtype == MemType_Normal &&
                s1_memattrs.inner.attrs != MemAttr_NC) then
                memattrs.inner.hints = s1_memattrs.inner.hints;
                memattrs.inner.transient = s1_memattrs.inner.transient;
            else
                memattrs.inner.hints = MemHint_RWA;
                memattrs.inner.transient = FALSE;

            if (s1_memattrs.memtype ==) &&
            (!s2_memattrs.inner.transient) &&
            (s2_memattrs.outer.attrs == MemType_NormalMemAttr_WB &&
                s1_memattrs.outer.attrs !=) &&
            (s2_memattrs.outer.hints == MemAttr_NC) then
                memattrs.outer.hints = s1_memattrs.outer.hints;
                memattrs.outer.transient = s1_memattrs.outer.transient;
            else
                memattrs.outer.hints = MemHint_RWA;
                memattrs.outer.transient = FALSE;

        else // Non-cacheable unless S1 is device
            if s1_memattrs.memtype == MemType_Device then
                memattrs = s1_memattrs;
            else
                MemAttrHints cacheability_attr;
                cacheability_attr.attrs = MemAttr_NC;

                memattrs.memtype = MemType_Normal;
                memattrs.inner = cacheability_attr;
                memattrs.outer = cacheability_attr;

        s2_shareability = DecodeShareability(s2_sh);
        memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

        memattrs.shareability = EffectiveShareability(memattrs);
        return memattrs;) &&

```

```
(!s2_memattrs.outer.transient)  
);
```

Library pseudocode for aarch64/

translation/vmsa\_tlbcontextvmsa\_memattr/AArch64.GetS1TLBContextAArch64.S2ApplyFWBMemAttr

```

// AArch64.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries
// AArch64.S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

TLBContextMemoryAttributes AArch64.GetS1TLBContext(AArch64.S2ApplyFWBMemAttrs(RegimeMemoryAttributes reg
bits(4) s2_attr, bits(2) s2_sh) SecurityStateMemoryAttributes

    if s2_attr<2> == '0' then // S2 Device, S1 any
        s2_device = TGxDecodeDevice tg)(s2_attr<1:0>);
        memattrs.memtype =
            TLBContextMemType_Device tlbcontext;

    case regime of
        when;
            if s1_memattrs.memtype == Regime_EL3MemType_Device tlbcontext = then
                memattrs.device = AArch64.TLBContextEL3S2CombineS1Device(ss, va, tg);
            when(s1_memattrs.device, s2_device);
            else
                memattrs.device = s2_device;

    elsif s2_attr<1:0> == '11' then // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elsif s2_attr<1:0> == '10' then // Force writeback
        memattrs.memtype = Regime_EL2MemType_Normal tlbcontext =;
        memattrs.inner.attrs = AArch64.TLBContextEL2MemAttr_WB(ss, va, tg);
        when;
        memattrs.outer.attrs = Regime_EL20MemAttr_WB tlbcontext =;

        if (s1_memattrs.memtype == AArch64.TLBContextEL20MemType_Normal(ss, va, tg);
        when&&
            s1_memattrs.inner.attrs != Regime_EL10MemAttr_NC tlbcontext =) then
                memattrs.inner.hints = s1_memattrs.inner.hints;
                memattrs.inner.transient = s1_memattrs.inner.transient;
            else
                memattrs.inner.hints = ;
                memattrs.inner.transient = FALSE;

            if (s1_memattrs.memtype == MemType_Normal &&
                s1_memattrs.outer.attrs != MemAttr_NC) then
                memattrs.outer.hints = s1_memattrs.outer.hints;
                memattrs.outer.transient = s1_memattrs.outer.transient;
            else
                memattrs.outer.hints = MemHint_RWA;
                memattrs.outer.transient = FALSE;

    else // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType_Device then
            memattrs = s1_memattrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr_NC;

            memattrs.memtype = MemType_Normal;
            memattrs.inner = cacheability_attr;
            memattrs.outer = cacheability_attr;

    s2 shareability = DecodeShareability(s2_sh);
    memattrs.shareability = S2CombinesS1Shareability(s1_memattrs.shareability, s2_shareability);
    memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

    memattrs.shareability = EffectiveShareabilityAArch64.TLBContextEL10MemHint_RWA(ss, va, tg);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;

```

```

return tlbcontext;{memattrs};
return memattrs;

```

## Library pseudocode for aarch64/translation/

vmsa\_tlbcontext/**AArch64.GetS2TLBContext****AArch64.GetS1TLBContext**

```

// AArch64.GetS2TLBContext()
// AArch64.GetS1TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(AArch64.GetS1TLBContext(Regime regime, SecurityState ss,ss, bits(64) v
assert(tg) EL2Enabled());

TLBContext tlbcontext;

tlbcontext.ss = ss;
tlbcontext.regime = case regime of
when Regime_EL10Regime_EL3;
tlbcontext.ipaspace = ipa.paspace;
tlbcontext.vmid =tlbcontext = VMIDAArch64.TLBContextEL3[];
tlbcontext.tg = tg;
tlbcontext.ia =(ss, va, tg);
when ZeroExtendRegime_EL2(ipa.address);
if tlbcontext = HaveCommonNotPrivateTransExtAArch64.TLBContextEL2() then
tlbcontext.cnp = if ipa.paspace ==(ss, va, tg);
when tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
when Regime_EL10 tlbcontext = AArch64.TLBContextEL10PAS_SecureRegime_EL20 then VSTTBR_EL2.CnP els
else
tlbcontext.cnp = '0';
(ss, va, tg);

tlbcontext.includes_s1 = FALSE;
tlbcontext.includes_s2 = TRUE;
tlbcontext.includes_s1 = TRUE;
// The following may be amended for EL1&0 Regime if caching of stage 2 is successful
tlbcontext.includes_s2 = FALSE;
return tlbcontext;

```

## Library pseudocode for aarch64/translation/

### vmsa\_tlbcontext/AArch64.TLBContextEL10AArch64.GetS2TLBContext

```
// AArch64.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime to match against TLB entries
// AArch64.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.TLBContextEL10(AArch64.GetS2TLBContext(SecurityState ss, bits(64) va, ss, FullAddress-1)
    assert
        EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid    = tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid    = VMID[];
    tlbcontext.asid    = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if tlbcontext.tg == tg;
    tlbcontext.ia = ZeroExtend(ipa.address);
    if HaveCommonNotPrivateTransExt() then
        if tlbcontext.cnp = if ipa.paspace == AArch64.GetVARangePAS_Secure(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    return tlbcontext;
```

## Library pseudocode for aarch64/translation/

vmsa\_tlbcontext/**AArch64.TLBContextEL2****AArch64.TLBContextEL10**

```
// AArch64.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries
// AArch64.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(AArch64.TLBContextEL10(SecurityState ss, bits(64) va, IGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime_EL2Regime_EL10;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp     = if tlbcontext.vmid == VMID[];
    tlbcontext.asid    = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER() then TTBR0_EL2.CnP else '0';
    then
        tlbcontext.cnp = TTBR0_EL1.CnP;
    else
        tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/

vmsa\_tlbcontext/**AArch64.TLBContextEL20****AArch64.TLBContextEL2**

```
// AArch64.TLBContextEL20()
// =====
// Gather translation context for accesses under EL20 regime to match against TLB entries
// AArch64.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(AArch64.TLBContextEL2(SecurityState ss, bits(64) va, IGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime_EL20Regime_EL2;
    tlbcontext.asid    = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if tlbcontext.cnp = if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL2.CnP;
        else
            tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';
    () then TTBR0_EL2.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/

**vmsa\_tlbcontext/AArch64.TLBContextEL3AArch64.TLBContextEL20**

```
// AArch64.TLBContextEL3()
// =====
// Gather translation context for accesses under EL3 regime to match against TLB entries
// AArch64.TLBContextEL20()
// =====
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(AArch64.TLBContextEL20(SecurityState ss, bits(64) va, IGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime_EL3Regime_EL20;
    tlbcontext.asid    = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp     = if
    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER() then TTBR0_EL3.CnP else '0';
    then
        tlbcontext.cnp = TTBR0_EL2.CnP;
    else
        tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/

**translation/vmsa\_translationvmsa\_tlbcontext/AArch64.AccessUsesELAArch64.TLBContextEL3**

```
// AArch64.AccessUsesEL()
// =====
// Returns the Exception Level of the regime that will manage the translation for a given access type.
// AArch64.TLBContextEL3()
// =====
// Gather translation context for accesses under EL3 regime to match against TLB entries

bits(2) TLBContext AArch64.AccessUsesEL(AArch64.TLBContextEL3(AccTypeSecurityState acctype)
    if acctype IN {ss, bits(64) va, AccType_UNPRIVIGx, tg} AccType_UNPRIVSTREAMTLBContext} then
        return tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = EL0Regime_EL3;
    elsif acctype == tlbcontext.tg == tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp     = if AccType_NV2REGISTERHaveCommonNotPrivateTransExt then
        return EL2;
    else
        return PSTATE.EL;() then TTBR0_EL3.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/

vmsa\_translation/**AArch64.FaultAllowsSetAccessFlag****AArch64.AccessUsesEL**

```
// AArch64.FaultAllowsSetAccessFlag()
// =====
// Determine whether the access flag could be set by HW given the fault status
// AArch64.AccessUsesEL()
// =====
// Returns the Exception Level of the regime that will manage the translation for a given access type.

boolean bits(2) AArch64.FaultAllowsSetAccessFlag(AArch64.AccessUsesEL(FaultRecordAccType fault)
    if fault.statuscode == acctype)
    if acctype IN { Fault_NoneAccType_UNPRIV } then
        return TRUE;
    elsif fault.statuscode IN {, Fault_AlignmentAccType_UNPRIVSTREAM, } then
        return Fault_PermissionEL0 } then
        return;
    elsif acctype == ConstrainUnpredictableAccType_NV2REGISTER( then
        return Unpredictable_AFUPDATEEL2) == Constraint_TRUE;
    else
        return FALSE; return PSTATE.EL;
```

## Library pseudocode for aarch64/translation/

vmsa\_translation/**AArch64.FullTranslate****AArch64.FaultAllowsSetAccessFlag**

```
// AArch64.FullTranslate()
// =====
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation
// AArch64.FaultAllowsSetAccessFlag()
// =====
// Determine whether the access flag could be set by HW given the fault status

AddressDescriptor boolean AArch64.FullTranslate(bits(64) va, AArch64.FaultAllowsSetAccessFlag(AccTypeFaultRecord fault)
    boolean iswrite, boolean aligned)

    fault = fault;
    if fault.statuscode == NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM});
    regime = TranslationRegime(PSTATE.EL, acctype);
    ss = SecurityStateAtEL(PSTATE.EL);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return return TRUE;
    elsif fault.statuscode IN { CreateFaultyAddressDescriptorFault_Alignment(va, fault);

    if regime ==, Regime_EL10Fault_Permission && } then
        return EL2EnabledConstrainUnpredictable() then
        slaarch64 = TRUE;
        s2fslwalk = FALSE;{
        AddressDescriptor Unpredictable_AFUPDATE pa;
        (fault, pa) == AArch64.S2TranslateConstraint_TRUE(fault, ipa, slaarch64, ss, s2fslwalk,
            acctype, aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(va, fault);
    else
        return pa;

;

    else
        return ipa; return FALSE;
```

Library pseudocode for aarch64/translation/  
vmsa\_translation/AArch64.MemSwapTableDescAArch64.FullTranslate

```

// AArch64.MemSwapTableDesc()
// =====
// Perform HW update of table descriptor as an atomic operation
// AArch64.FullTranslate()
// =====
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

(FaultRecord, bits(64))AddressDescriptor AArch64.MemSwapTableDesc(AArch64.FullTranslate(bits(64) va, FaultRecord,
                                                                    bits(64) new_desc, bit ee, acctype,
                                                                    boolean iswrite, boolean aligned))

    fault =
                                                                    AddressDescriptorNoFault descupdateaddress)

    descupdateaccess = ();
    fault.acctype = acctype;
    fault.write = iswrite;

    ispriv = PSTATE.EL != CreateAccessDescriptorEL0(&& !(acctype IN {AccType_ATOMICRW, AccType_UNPRIV});,
    FaultRecordAccType_UNPRIVSTREAM fault = fault_in;
    boolean iswrite;

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    (memstatus, mem_desc) = {};
    regime = PhysMemReadTranslationRegime(descupdateaddress, 8, descupdateaccess);
    if ee == '1' then
        mem_desc = (PSTATE.EL, acctype);
    ss = BigEndianReverseSecurityStateAtEL(mem_desc);

    if(PSTATE.EL); IsFaultAddressDescriptor(memstatus) then
        iswrite = FALSE;
        fault = ipa;
        (fault, ipa) = HandleExternalTTWAbortAArch64.S1Translate(memstatus, iswrite, descupdateaddress, descupdateaccess,
                                                                    8, fault);
        if(fault, regime, ss, va, acctype, aligned, iswrite, ispriv);

    if fault.statuscode != IsFaultFault_None(fault.statuscode) then
        fault.acctype = then
        return AccType_ATOMICRWCreateFaultyAddressDescriptor;
        return (fault, bits(64) UNKNOWN);
(va, fault);

    if mem_desc == prev_desc then
        ordered_new_desc = if ee == '1' then if regime == BigEndianReverseRegime_EL10(new_desc) else r
        memstatus = && PhysMemWriteEL2Enabled(descupdateaddress, 8, descupdateaccess, ordered_new_desc);

        if() then
        slaarch64 = TRUE;
        s2fslwalk = FALSE; IsFaultAddressDescriptor(memstatus) then
            iswrite = TRUE;
            fault = pa;
            (fault, pa) = HandleExternalTTWAbortAArch64.S2Translate(memstatus, iswrite, descupdateaddress, descupdateaccess,
                                                                    8, fault);
            fault.acctype = memstatus.acctype;
            if(fault, ipa, slaarch64, ss, s2fslwalk,
                                                                    acctype, aligned, iswrite, ispriv);

        if fault.statuscode != IsFaultFault_None(fault.statuscode) then
            fault.acctype = then
            return AccType_ATOMICRWCreateFaultyAddressDescriptor;
            return (fault, bits(64) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;

    return (fault, mem_desc);(va, fault);
    else
        return pa;

```

```
else  
    return ipa;
```

Library pseudocode for aarch64/translation/  
vmsa\_translation/AArch64.S1DisabledOutputAArch64.MemSwapTableDesc

```

// AArch64.S1DisabledOutput()
// AArch64.MemSwapTableDesc()
// =====
// Map the the VA to IPA/PA and assign default memory attributes
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, AddressDescriptor)(FaultRecord, bits(64)) AArch64.S1DisabledOutput(AArch64.MemSwapTableDesc
bits(64) new_desc, bit ee, RegimeAddressDescriptor regime

descupdateaccess =

SecurityStateCreateAccessDescriptor ss, bits(64)
AccTypeAccType_ATOMICRW acctype, boolean align

walkparams =); AArch64.GetS1TTWParams(regime, va);
FaultRecord fault = fault_in;
boolean iswrite;

// No memory page is guarded when stage 1 address translation is disabled // All observers in the
// following memory read and write accesses atomically.
(memstatus, mem_desc) =
SetInGuardedPagePhysMemRead(FALSE);

// Output Address(descupdateaddress, 8, descupdateaccess);
if ee == '1' then
    mem_desc =
    FullAddressBigEndianReverse oa;
    oa.address = va<51:0>;
    case ss of
        when(mem_desc);

    if SS_SecureIsFault oa.paspace =(memstatus) then
        iswrite = FALSE;
        fault = PAS_SecureHandleExternalTTWAbort;
        when(memstatus, iswrite, descupdateaddress, descupdateaccess,
            8, fault);
        if SS_NonSecureIsFault oa.paspace =(fault.statuscode) then
            fault.acctype = PAS_NonSecureAccType_ATOMICRW;;
            return (fault, bits(64) UNKNOWN);

    if mem_desc == prev_desc then
        ordered_new_desc = if ee == '1' then

        MemoryAttributesBigEndianReverse memattrs;
        if regime ==(new_desc) else new_desc;
        memstatus = Regime_EL10PhysMemWrite &&(descupdateaddress, 8, descupdateaccess, ordered_new_desc);

        if EL2EnabledIsFault() && walkparams.dc == '1' then(memstatus) then
            iswrite = TRUE;
            fault =
            MemAttrHintsHandleExternalTTWAbort default_cacheability;
            default_cacheability.attrs =(memstatus, iswrite, descupdateaddress, descupdateaccess,
                8, fault);
            fault.acctype = memstatus.acctype;
            if MemAttr_WBIsFault;
            default_cacheability.hints =(fault.statuscode) then
                fault.acctype = MemHint_RWAAccType_ATOMICRW;
            default_cacheability.transient = FALSE;

            memattrs.memtype = MemType_Normal;
            memattrs.outer = default_cacheability;
            memattrs.inner = default_cacheability;
            memattrs.shareability = Shareability_NSH;
            memattrs.tagged = walkparams.dct == '1';
            memattrs.xs = '0';
        elseif acctype == AccType_IFETCH then
            MemAttrHints i_cache_attr;
            if AArch64.S1ICacheEnabled(regime) then
                i_cache_attr.attrs = MemAttr_WT;
                i_cache_attr.hints = MemHint_RA;
                i_cache_attr.transient = FALSE;
            else

```

```

        i_cache_attr.attrs      = MemAttr_NC;

        memattrs.memtype        = MemType_Normal;
        memattrs.outer          = i_cache_attr;
        memattrs.inner          = i_cache_attr;
        memattrs.shareability   = Shareability_OSH;
        memattrs.tagged         = FALSE;
        memattrs.xs             = '1';
    else
        memattrs.memtype        = MemType_Device;
        memattrs.device          = DeviceType_nGnRnE;
        memattrs.shareability    = Shareability_OSH;
        memattrs.xs             = '1';

    fault.level = 0;
    addrtop     = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
    if !IsZero(va<addrtop:AArch64.PAMax()>) then
        fault.statuscode = Fault_AddressSize;
    elseif AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid, memattrs) then
        fault.statuscode = Fault_Alignment;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va, oa, memattrs);
        return (fault, ipa);
    return (fault, bits(64) UNKNOWN);

    // Reflect what is now in memory (in little endian format)
    mem_desc = new_desc;

    return (fault, mem_desc);

```

Library pseudocode for aarch64/translation/  
vmsa\_translation/AArch64.S1TranslateAArch64.S1DisabledOutput

```

// AArch64.S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime
// AArch64.S1DisabledOutput()
// =====
// Map the the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1Translate(AArch64.S1DisabledOutput(FaultRecord fault_in, Regime
SecurityState ss, bits(64) va,
AccType acctype, boolean aligned_in,
boolean iswrite_in, boolean ispriv)acctype, boolean

walkparams =
FaultRecord fault = fault_in;
boolean aligned = aligned_in;
boolean iswrite = iswrite_in;
// Prepare fault fields in case a fault is detected
fault.secondstage = FALSE;
fault.s2fslwalk = FALSE;

if !AArch64.S1Enabled(regime) then
return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

walkparams = AArch64.GetS1TTWParams(regime, va);

if ((regime, va); AArch64.S1InvalidTxSZFaultRecord(walkparams) ||
(!ispriv && walkparams.e0pd == '1') || fault = fault_in;

// No memory page is guarded when stage 1 address translation is disabled
AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
fault.statuscode = Fault_Translation;
fault.level = 0;
return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
(fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime,
ss, acctype, iswrite, ispriv);

if fault.statuscode != Fault_None then
return (fault, AddressDescriptor UNKNOWN);

if acctype == AccType_IFETCH then
// Flag the fetched instruction is from a guarded page
SetInGuardedPage(walkstate.guardedpage == '1');
(FALSE);

if // Output Address AArch64.S1HasAlignmentFaultFullAddress(acctype, aligned, walkparams.ntlsr
walkstate.memattrs) then
fault.statuscode = oa;
oa.address = va<51:0>;
case ss of
when Fault_AlignmentSS_Secure;
elseif oa.paspace = IsAtomicRWPAS_Secure(acctype) then
if;
when AArch64.S1HasPermissionsFaultSS_NonSecure(regime, ss, walkstate, walkparams,
ispriv, acctype, FALSE) then
// The Permission fault was not caused by lack of write permissions
fault.statuscode = oa.paspace = Fault_PermissionPAS_NonSecure;
fault.write = FALSE;
elseif; AArch64.S1HasPermissionsFaultMemoryAttributes(regime, ss, walkstate, walkparams,
ispriv, acctype, TRUE) then
// The Permission fault was caused by lack of write permissions
fault.statuscode = memattrs;
if regime == Fault_Permission;
fault.write = TRUE;

```

```

elseif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                     ispriv, acctype, iswrite) then
    fault.statuscode = Fault_Permission;

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == Fault_None &&
        walkparams.ha == '1' &&
        walkparams.hd == '1' &&
        descriptor<51> == '1' && // Descriptor DBM bit
        (IsAtomicRW(acctype) || iswrite) &&
        !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
        // Clear descriptor AP[2] bit permitting stage 1 writes
        new_desc<7> = '0';

    AddressDescriptor descupdateaddress;
    FaultRecord s2fault;
    // Either the access flag was clear or AP<2> is set
    if new_desc != descriptor then
        if regime == Regime_EL10 && EL2Enabled() then
            slaarch64 = TRUE;
            s2fslwalk = TRUE;
            aligned = TRUE;
            iswrite = TRUE;
            (s2fault, descupdateaddress) = () && walkparams.dc == '1' then AArch64.S2TranslateMemAttr(
                ss, s2fslwalk, default_cacheability,
                default_cacheability.attrs = AccType_ATOMICRWMemAttr_WB,
                aligned, iswrite, ispriv);

            if s2fault.statuscode !=;
            default_cacheability.hints = Fault_NoneMemHint_RWA then
                return (s2fault,;
            default_cacheability.transient = FALSE;

            memattrs.memtype = AddressDescriptorMemType_Normal UNKNOWN);
            else
                descupdateaddress = descaddress;

            (fault, mem_desc) =;
            memattrs.outer = default_cacheability;
            memattrs.inner = default_cacheability;
            memattrs.shareability = AArch64.MemSwapTableDescShareability_NSH(fault, descriptor, new_desc,
                walkparams.ee, descupdateaddress);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode !=;
    memattrs.tagged = walkparams.dct == '1';
    memattrs.xs = '0';
    elseif acctype == Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    // Output Address
    oa = Stage0A(va, walkparams.tgx, walkstate);
    MemoryAttributes memattrs;
    if (acctype == AccType_IFETCH &&
        (walkstate.memattrs.memtype == then MemType_DeviceMemAttrHints || !i_cache_attr;
        if AArch64.S1ICacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = (regime) then
            i_cache_attr.attrs = NormalNCMemAttrMemAttr_WT();
        memattrs.xs = walkstate.memattrs.xs;
    elseif (acctype !=;

```

```

        i_cache_attr.hints = AccType_IFETCHMemHint_RA && !;
        i_cache_attr.transient = FALSE;
    else
        i_cache_attr.attrs = AArch64.S1DCacheEnabledMemAttr_NC(regime) &&
            walkstate.memattrs.memtype ==;

        memattrs.memtype = MemType_Normal) then
            // Treat memory attributes as Normal Non-Cacheable
            memattrs =;
            memattrs.outer = i_cache_attr;
            memattrs.inner = i_cache_attr;
            memattrs.shareability = NormalNCMemAttrShareability_OSH();
            memattrs.xs = walkstate.memattrs.xs;

            // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
            // on the Tagged attribute
            if;
                memattrs.tagged = FALSE;
                memattrs.xs = '1';
            else
                memattrs.memtype = HaveMTE2ExtMemType_Device() && walkstate.memattrs.tagged then
                    memattrs.tagged =;
                    memattrs.device = ConstrainUnpredictableBoolDeviceType_nGnRnE(;
                    memattrs.shareability = Unpredictable_S1CTAGGEDShareability_OSH);
                else
                    memattrs = walkstate.memattrs;
;
        memattrs.xs = '1';

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == fault.level = 0;
        addrtop = Regime_EL10AArch64.AddrTop && (walkparams.tbid, acctype, walkparams.tbi);
        if ! EL2EnabledIsZero() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION DEFINED "Apply effective shareability at stage 1")) then
            memattrs.shareability = walkstate.memattrs.shareability;
        else
            memattrs.shareability = (va < addrtop: EffectiveShareabilityAArch64.PAMax(memattrs);

    if acctype == (>) then
        fault.statuscode = AccType_ATOMICS64Fault_AddressSize && memattrs.memtype ==;
    elsif MemType_NormalAArch64.S1HasAlignmentFault then
        if memattrs.inner.attrs != (acctype, aligned, walkparams.ntlsmid, memattrs) then
            fault.statuscode = MemAttr_NCFault_Alignment || memattrs.outer.attrs !=;

    if fault.statuscode != MemAttr_NCFault_None then
        fault.statuscode = Fault_Exclusive;
        return (fault, then
        return (fault, AddressDescriptor UNKNOWN);

    ipa = else
        ipa = CreateAddressDescriptor(va, oa, memattrs);
    return (fault, ipa);

```

Library pseudocode for aarch64/translation/  
vmsa\_translation/AArch64.S2TranslateAArch64.S1Translate

```

// AArch64.S2Translate()
// AArch64.S1Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S2Translate(AArch64.S1Translate(FaultRecord fault_in, AddressDescriptor descaddress,
                                                                    boolean slaarch64, regime, SecurityState ss,
                                                                    boolean s2fslwalk, ss, bits(64) va, AccType acctype,
                                                                    boolean aligned, boolean iswrite,
                                                                    boolean ispriv)
    walkparams = acctype, boolean aligned_in,
                                                                    boolean iswrite_in, boolean ispriv) AArch64.GetS2TTWParams(walkparams,
                                                                    boolean iswrite_in, boolean ispriv)

    FaultRecord fault = fault_in;
    boolean aligned = aligned_in;
    boolean iswrite = iswrite_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    // Prepare fault fields in case a fault is detected
    fault.statuscode = if ! Fault_None AArch64.S1Enabled; // Ignore any faults from stage 1
    fault.secondstage = TRUE;
    fault.s2fslwalk = s2fslwalk;
    fault.ipaddress = ipa.address;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        return (fault, ipa);

    if ((regime) then
        return AArch64.S2InvalidTxSZ AArch64.S1DisabledOutput(walkparams, slaarch64) || (fault, regime, ss,
                                                                    walkparams =
                                                                    AArch64.S2InvalidSL AArch64.GetS1TTWParams(walkparams) || (regime, va);

    if (
        AArch64.S2InconsistentSL AArch64.S1InvalidTxSZ(walkparams) || (walkparams) ||
        (!ispriv && walkparams.e0pd == '1') ||
        AArch64.IPAIsOutOfRange AArch64.VAIsOutOfRange(ipa.address.address, walkparams)) then
    (va, acctype, regime, walkparams)) then
        fault.statuscode = Fault_Translation;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descaddress;
    TTWState walkstate;
    bits(64) descriptor;
    bits(64) new_desc;
    bits(64) mem_desc;
    repeat
        (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk AArch64.S1Walk(fault, ipa, walkparams,
                                                                    acctype, iswrite, slaarch64);
    (fault, walkparams, va, regime,
                                                                    ss, acctype, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if if acctype == AArch64.S2HasAlignmentFault AccType_IFETCH(acctype, aligned, walkstate.memattrs) then
        fault.statuscode = then
        // Flag the fetched instruction is from a guarded page SetInGuardedPage(walkstate.guardedpage,
                                                                    fault.statuscode = then
        if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid,
                                                                    walkstate.memattrs) then
            fault.statuscode = Fault_Alignment;
        elsif IsAtomicRW(acctype) then
            if AArch64.S2HasPermissionsFault AArch64.S1HasPermissionsFault(s2fslwalk, walkstate, ss, walkp
                                                                    (regime, ss, walkstate, walkparams,
                                                                    ispriv, acctype, FALSE) then

```

```

        // The Permission fault was not caused by lack of write permissions
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
        elseif AArch64.S2HasPermissionsFaultAArch64.S1HasPermissionsFault(s2fslwalk, walkstate, ss, wa
(regime, ss, walkstate, walkparams,
                                ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions.
        // However, HW updates, which are atomic writes for stage 1
        // descriptors, permissions fault reflect the original access.
        // The permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        if !fault.s2fslwalk then
            fault.write = TRUE;
            fault.write = TRUE;
        elseif AArch64.S2HasPermissionsFaultAArch64.S1HasPermissionsFault(s2fslwalk, walkstate, ss, walkpa
(regime, ss, walkstate, walkparams,
                                ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;

        new_desc = descriptor;
        if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
            // Set descriptor AF bit
            new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permissions
        // will already reflect a configuration permitting writes.
        // The update of the descriptor occurs only if the descriptor bits in
        // memory do not reflect that and the access instigates a write.
        if (fault.statuscode == Fault_None &&
            walkparams.ha == '1' &&
            walkparams.hd == '1' &&
            descriptor<51> == '1' && // Descriptor DBM bit
            (IsAtomicRW(acctype) || iswrite) &&
            !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
            // Set descriptor S2AP[1] bit permitting stage 2 writes
            new_desc<7> = '1';

        // Either the access flag was clear or S2AP<1> is clear
        if new_desc != descriptor then
            (fault, mem_desc) = // Clear descriptor AP[2] bit permitting stage 1 writes
                new_desc<7> = '0'; AddressDescriptor descupdateaddress;
            FaultRecord s2fault;
        // Either the access flag was clear or AP<2> is set
        if new_desc != descriptor then
            if regime == Regime_EL10 && EL2Enabled() then
                slaarch64 = TRUE;
                s2fslwalk = TRUE;
                aligned    = TRUE;
                iswrite    = TRUE;
                (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress, slaarch64,
                                                                    ss, s2fslwalk, AccType_ATOMICRW,
                                                                    aligned, iswrite, ispriv);

                if s2fault.statuscode != Fault_None then
                    return (s2fault, AddressDescriptor UNKNOWN);
                else
                    descupdateaddress = descaddress;

            (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                            walkparams.ee, descaddress);
                                                            walkparams.ee, descupdateaddress);

        until new_desc == descriptor || mem_desc == new_desc;

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);

        ipa_64 = // Output Address
        oa = ZeroExtend(ipa.paddress.address, 64);

```

```

// Output Address
oa = StageOA(ipa_64, walkparams.tgx, walkstate);(va, walkparams.tgx, walkstate);
MemoryAttributes s2_memattrs;
if ((s2fslwalk &&
    walkstate.memattrs.memtype == memattrs;
if (acctype == MemType_Device && walkparams.ptw == '0') ||
    (acctype == AccType_IFETCH &&
        (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
    (acctype != || ! AArch64.S1ICacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
elseif (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == && ! AArch64.S1DCacheEnabled(regime) &&
    walkstate.memattrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;
    memattrs.xs = walkstate.memattrs.xs;

if !s2fslwalk && acctype == // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
    // on the Tagged attribute
    if HaveMTE2Ext() && walkstate.memattrs.tagged then
        memattrs.tagged = ConstrainUnpredictableBool(Unpredictable_S1CTAGGED);
    else
        memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

if acctype == AccType_ATOMICLS64 && s2_memattrs.memtype == && memattrs.memtype == MemType_Normal then
    if s2_memattrs.inner.attrs != if memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.a
        fault.statuscode = Fault_Exclusive;
        return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if walkparams.fwb == '0' then
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
else
    memattrs = s2_memattrs;
UNKNOWN);

pa = ipa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
return (fault, pa);(va, oa, memattrs);
return (fault, ipa);

```

Library pseudocode for aarch64/translation/  
vmsa\_translation/AArch64.TranslateAddressAArch64.S2Translate

```

// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address
// AArch64.S2Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes

AddressDescriptor(FaultRecord, AddressDescriptor) AArch64.TranslateAddress(bits(64) va, AArch64.S2Translate(
    boolean slaarch64, SecurityState ss,
    boolean s2fslwalk, AccType acctype, boolean iswrite,
    boolean aligned, integer size)
    result = acctype,
    boolean aligned, boolean iswrite,
    boolean ispriv)
walkparams = AArch64.FullTranslateAArch64.GetS2TTWParams(va, acctype, iswrite, aligned);

if !(ss, ipa.paddress.paspace, slaarch64); IsFaultFaultRecord(result) then
    result.fault = fault = fault_in;

// Prepare fault fields in case a fault is detected
fault.statuscode = AArch64.CheckDebugFault_None(va, acctype, iswrite, size);
; // Ignore any faults from stage 1
fault.secondstage = TRUE;
fault.s2fslwalk = s2fslwalk;
fault.ipaddress = ipa.paddress;

// Update virtual address for abort functions
result.vaddress = if walkparams.vm != '1' then
    // Stage 2 translation is disabled
    return (fault, ipa);

if ( AArch64.S2InvalidTxSZ(walkparams, slaarch64) ||
    AArch64.S2InvalidSL(walkparams) ||
    AArch64.S2InconsistentSL(walkparams) ||
    AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
    (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams, ss,
        acctype, iswrite, slaarch64);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    if AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, ss, walkparams,
        ispriv, acctype, FALSE) then
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, ss, walkparams,
        ispriv, acctype, TRUE) then
        // The permission fault was caused by lack of write permissions.
        // However, HW updates, which are atomic writes for stage 1
        // descriptors, permissions fault reflect the original access.
        fault.statuscode = Fault_Permission;
        if !fault.s2fslwalk then
            fault.write = TRUE;
    elseif AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, ss, walkparams,
        ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

```

```

new_desc = descriptor;
if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
    // Set descriptor AF bit
    new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == Fault_None &&
        walkparams.ha == '1' &&
        walkparams.hd == '1' &&
        descriptor<51> == '1' && // Descriptor DBM bit
        (IsAtomicRW(acctype) || iswrite) &&
        !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
        // Set descriptor S2AP[1] bit permitting stage 2 writes
        new_desc<7> = '1';

    // Either the access flag was clear or S2AP<1> is clear
    if new_desc != descriptor then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                    walkparams.ee, descaddress);

until new_desc == descriptor || mem_desc == new_desc;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

ipa_64 = ZeroExtend(ipa.paddress.address, 64);
// Output Address
oa = Stage0A(ipa_64, walkparams.tgx, walkstate);
MemoryAttributes s2_memattrs;
if ((s2fslwalk &&
    walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
    (acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
    (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

if !s2fslwalk && acctype == AccType_ATOMICS64 && s2_memattrs.memtype == MemType_Normal then
    if s2_memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.attrs != MemAttr_NC then
        fault.statuscode = Fault_Exclusive;
        return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if walkparams.fwb == '0' then
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
else
    memattrs = s2_memattrs;

pa = CreateAddressDescriptor(va);

return result;(ipa.vaddress, oa, memattrs);
return (fault, pa);

```

## Library pseudocode for aarch64/

translation/vmsa\_tentry/vmsa\_translation/AArch64.BlockDescSupportedAArch64.TranslateAddress

```
// AArch64.BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule size
// and level
// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

boolean AddressDescriptor AArch64.BlockDescSupported(bit ds, AArch64.TranslateAddress(bits(64) va, TGxAccType
    case tgx of
        when acctype, boolean iswrite,
            boolean aligned, integer size)

    result = TGx_4KBAArch64.FullTranslate return level == 2 || level == 1 || (level == 0 && ds == '1');
    when(va, acctype, iswrite, aligned);

    if ! TGx_16KBIsFault return level == 2 || (level == 1 && ds == '1');
    when(result) then
        result.fault = TGx_64KBAArch64.CheckDebug return level == 2 || (level == 1 && (va, acctype, iswrite, aligned));

    // Update virtual address for abort functions
    result.vaddress = AArch64.PAMaxZeroExtend() == 52);
(va);

    return FALSE; return result;
```

## Library pseudocode for aarch64/translation/

vmsa\_tentry/AArch64.BlocknTFaultsAArch64.BlockDescSupported

```
// AArch64.BlocknTFaults()
// =====
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault
// AArch64.BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlocknTFaults(bits(64) descriptor)
    if !AArch64.BlockDescSupported(bit ds, HaveBlockBBMTGx() then
        return FALSE;

    bbm_level = tgx, integer level)
    case tgx of
        when AArch64.BlockBBMSupportLevelTGx_4KB();
        nT_faults = boolean IMPLEMENTATION_DEFINED "BBM level 1 or 2 support nT bit causes Translation return
            when return level == 2 || (level == 1 && ds == '1');
            when TGx_64KB return level == 2 || (level == 1 && AArch64.PAMaxFaultTGx_16KB";
    () == 52);

    return bbm_level IN {1, 2} && descriptor<16> == '1' && nT_faults; return FALSE;
```

**Library pseudocode for aarch64/translation/  
vmsa\_ttentry/AArch64.ContiguousBitAArch64.BlocknTFaults**

```
// AArch64.ContiguousBit()
// AArch64.BlocknTFaults()
// =====
// Get the value of the contiguous bit
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

bitboolean AArch64.ContiguousBit(AArch64.BlocknTFaults(bits(64) descriptor)
    if !TGxHaveBlockBBM tgx, integer level, bits(64) descriptor)
    if tgx ==() then
        return FALSE;

    bbm_level = TGx_64KBAArch64.BlockBBMSupportLevel && level == 1 && !();
    nT_faults = boolean IMPLEMENTATION_DEFINED "BBM level 1 or 2 support nT bit causes TranslationHave52B
        return '0'; // RES0
    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0
    if tgx == TGx_4KB && level == 0 then
        return '0'; // RES0
    "
    return descriptor<52>; return bbm_level IN {1, 2} && descriptor<16> == '1' && nT_faults;
```

**Library pseudocode for aarch64/translation/  
vmsa\_ttentry/AArch64.DecodeDescriptorTypeAArch64.ContiguousBit**

```
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table
// AArch64.ContiguousBit()
// =====
// Get the value of the contiguous bit

DescriptorTypebit AArch64.DecodeDescriptorType(bits(64) descriptor, bit ds, AArch64.ContiguousBit(
    TGx tgx, integer level)
    if descriptor<1:0> == '11' && level ==tgx, integer level, bits(64) descriptor)
    if tgx == FINAL_LEVELTGx_64KB then
        return&& level == 1 && ! DescriptorType_PageHave52BitVAExt;
    elseif descriptor<1:0> == '11' then
        return() then
        return '0'; // RES0
    if tgx == DescriptorType_TableTGx_16KB;
    elseif descriptor<1:0> == '01' then
        if&& level == 1 then
            return '0'; // RES0
    if tgx == AArch64.BlockDescSupportedTGx_4KB(ds, tgx, level) then
        return DescriptorType_Block;
    else
        return DescriptorType_Invalid;
    else
        return DescriptorType_Invalid;&& level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

## Library pseudocode for aarch64/translation/ vmsa\_tentry/AArch64.S1ApplyOutputPermsAArch64.DecodeDescriptorType

```
// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

PermissionsDescriptorType AArch64.S1ApplyOutputPerms(AArch64.DecodeDescriptorType(bits(64) descriptor, b
    if descriptor<1:0> == '11' && level ==
        RegimeFINAL_LEVEL regime, then
        return S1TTWParamsDescriptorType_Page walkparams);
    elsif descriptor<1:0> == '11' then
        return
        PermissionsDescriptorType_Table permissions = permissions_in;
        if regime ==;
        elsif descriptor<1:0> == '01' then
            if Regime_EL10AArch64.BlockDescSupported &&(ds, tdx, level) then
                return EL2EnabledDescriptorType_Block() && walkparams.nv1 == '1' then
                    permissions.ap<2:1> = descriptor<7>:'0';
                    permissions.pxn = descriptor<54>;
            elsif;
            else
                return ;
        else
            return DescriptorType_InvalidHasUnprivilegedDescriptorType_Invalid(regime) then
                permissions.ap<2:1> = descriptor<7:6>;
                permissions.uxn = descriptor<54>;
                permissions.pxn = descriptor<53>;
            else
                permissions.ap<2:1> = descriptor<7>:'1';
                permissions.xn = descriptor<54>;

        // Descriptors marked with DBM set have the effective value of AP[2] cleared.
        // This implies no Permission faults caused by lack of write permissions are
        // reported, and the Dirty bit can be set.
        if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
            permissions.ap<2> = '0';

        return permissions;

```

## Library pseudocode for aarch64/translation/

### vmsa\_tentry/AArch64.S1ApplyTablePermsAArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors
// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyTablePerms(AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(64) des
    Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table = descriptor<62>:'0';
        pxn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

        permissions.ap<2:1> = descriptor<7>:'0';
        permissions.pxn = descriptor<54>;
    elseif HasUnprivileged(regime) then
        ap_table = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn = descriptor<54>;
        permissions.pxn = descriptor<53>;
    else
        ap_table = descriptor<62>:'0';
        xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;
        permissions.ap<2:1> = descriptor<7>:'1';
        permissions.xn = descriptor<54>;

    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.ap<2> = '0';

    return permissions;
```

## Library pseudocode for aarch64/translation/ vmsa\_ttentry/AArch64.S2ApplyOutputPermsAArch64.S1ApplyTablePerms

```
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors
// AArch64.S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(64) descriptor, AArch64.S1ApplyTablePerms( S2TTWParamsPermissions
Regime regime, S1TTWParams walkparams)
Permissions permissions;

permissions.s2ap = descriptor<7:6>;
permissions.s2xn = descriptor<54>;

if permissions = permissions_in;
bits(2) ap_table;
bit pxn_table;
bit uxn_table;
bit xn_table;
if regime == && EL2Enabled() && walkparams.nv1 == '1' then
    ap_table = descriptor<62>:'0';
    pxn_table = descriptor<60>;
    permissions.ap_table = permissions.ap_table OR ap_table;
    permissions.pxn_table = permissions.pxn_table OR pxn_table;

elseif HasUnprivilegedHaveExtendedExecuteNeverExtRegime_EL10() then
    permissions.s2xnx = descriptor<53>;
(regime) then
    ap_table = descriptor<62:61>;
    uxn_table = descriptor<60>;
    pxn_table = descriptor<59>;
    permissions.ap_table = permissions.ap_table OR ap_table;
    permissions.uxn_table = permissions.uxn_table OR uxn_table;
    permissions.pxn_table = permissions.pxn_table OR pxn_table;
else
    permissions.s2xnx = '0';

// Descriptors marked with DBM set have the effective value of S2AP[1] set.
// This implies no Permission faults caused by lack of write permissions are
// reported, and the Dirty bit can be set.
if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
    permissions.s2ap<1> = '1';
    ap_table = descriptor<62>:'0';
    xn_table = descriptor<60>;
    permissions.ap_table = permissions.ap_table OR ap_table;
    permissions.xn_table = permissions.xn_table OR xn_table;

return permissions;
```

## Library pseudocode for aarch64/

translation/~~vm~~sa\_walk~~vm~~sa\_ttentry/~~AArch64.S1InitialTTWState~~~~AArch64.S2ApplyOutputPerms~~

```
// AArch64.S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

TTWStatePermissions AArch64.S1InitialTTWState(AArch64.S2ApplyOutputPerms(bits(64) descriptor, S1TTWParams,
SecurityState ss)

    TTWState walkstate;
    FullAddress tablebase;
    Permissions permissions;

    startlevel = permissions.s2ap = descriptor<7:6>;
    permissions.s2xn = descriptor<54>;

    if AArch64.S1StartLevelHaveExtendedExecuteNeverExt(walkparams);
        ttbr = AArch64.S1TTBR(regime, va);
        case ss of
            when SS_Secure tablebase.paspace = PAS_Secure;
            when SS_NonSecure tablebase.paspace = PAS_NonSecure;

        tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
            walkparams.tgx, startlevel);

        permissions.ap_table = Zeros();
        if HasUnprivileged(regime) then
            permissions.uxn_table = Zeros();
            permissions.pxn_table = Zeros();
        else
            permissions.xn_table = Zeros();

        walkstate.baseaddress = tablebase;
        walkstate.level = startlevel;
        walkstate.istable = TRUE;
        // In regimes that support global and non-global translations, translation
        // table entries from lookup levels other than the final level of lookup
        // are treated as being non-global
        walkstate.nG = if HasUnprivileged(regime) then '1' else '0';
        walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
        walkstate.permissions = permissions;
    () then
        permissions.s2xnx = descriptor<53>;
    else
        permissions.s2xnx = '0';

    return walkstate; // Descriptors marked with DBM set have the effective value of S2AP[1] set.
    // This implies no permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.s2ap<1> = '1';

    return permissions;
```

Library pseudocode for aarch64/translation/  
vmsa\_walk/~~AArch64.S1NextWalkStateLast~~~~AArch64.S1InitialTTWState~~

```

// AArch64.S1NextWalkStateLast()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation
// AArch64.S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1NextWalkStateLast(AArch64.S1InitialTTWState(TTWStateS1TTWParams currentstate, walkparams
                                S1TTWParams walkparams, bits(64) descriptor)

    TTWState    nextstate; walkstate;
    FullAddress baseaddress;

    if currentstate.level == tablebase; FINAL_LEVELPermissions then
        baseaddress.address = permissions;

    startlevel = AArch64.PageBaseAArch64.S1StartLevel(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = (walkparams);
    ttbr = AArch64.BlockBaseAArch64.S1TTBR(descriptor, walkparams.ds, walkparams.tgx,
                                           currentstate.level);

    if currentstate.baseaddress.paspace == (regime, va);
    case ss of
        when PAS_SecureSS_Secure then
            // Determine PA space of the block from NS bit
            baseaddress.paspace = if descriptor<5> == '0' then tablebase.paspace = PAS_Secure else;
        when PAS_NonSecureSS_NonSecure;
    else
        baseaddress.paspace = tablebase.paspace = PAS_NonSecure;

    nextstate.istable = FALSE;
    nextstate.level = currentstate.level;
    nextstate.baseaddress = baseaddress;

    attrindx = descriptor<4:2>;
    sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    attr = tablebase.address = MAIRAttrAArch64.TTBaseAddress((ttbr, walkparams.txsz, walkparams.ps, wa
                                walkparams.tgx, startlevel);

    permissions.ap_table = UIntZeros(attrindx), walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs = ();
    if S1DecodeMemAttrHasUnprivileged(attr, sh, slaarch64);
    nextstate.permissions = (regime) then
        permissions.uxn_table = AArch64.S1ApplyOutputPermsZeros(currentstate.permissions, descriptor,
                                regime, walkparams);

    nextstate.contiguous = ();
    permissions.pxn_table = AArch64.ContiguousBitZeros(walkparams.tgx, currentstate.level, descriptor

    if !();
    else
        permissions.xn_table = Zeros();

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG = if HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif ss == (regime) then '1' else '0';
    walkstate.memattrs = SS_SecureWalkMemAttr && currentstate.baseaddress.paspace == PAS_NonSecure th
        // In Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
    nextstate.nG = '1';
    else
        nextstate.nG = descriptor<11>;

```

```
(walkparams.sh, walkparams.irgn, walkparams.orgn);  
walkstate.permissions = permissions;  
  
nextstate.guardedpage = descriptor<50>;  
  
return nextstate; return walkstate;
```

Library pseudocode for aarch64/translation/  
vmsa\_walk/AArch64.S1NextWalkStateTableAArch64.S1NextWalkStateLast

```

// AArch64.S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level
// AArch64.S1NextWalkStateLast()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateTable(AArch64.S1NextWalkStateLast(TTWState currentstate, Regime regime, S
                                S1TTWParams walkparams,
                                bits(64) descriptor)

    TTWState    nextstate;
    FullAddress tablebase;
baseaddress;

    tablebase.address = if currentstate.level == AArch64.NextTableBaseFINAL_LEVEL(descriptor, walkpara
    if currentstate.baseaddress.paspace == then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx,
                                currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        tablebase.paspace = if descriptor<63> == '0' then // Determine PA space of the block from
        baseaddress.paspace = if descriptor<5> == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable = TRUE;
    nextstate.nG      = currentstate.nG;
    nextstate.level   = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs = currentstate.memattrs;

    if walkparams.hpd == '0' then
        nextstate.permissions = baseaddress.paspace = ;

    nextstate.istable = FALSE;
    nextstate.level   = currentstate.level;
    nextstate.baseaddress = baseaddress;

    attrindx = descriptor<4:2>;
    sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    attr = MAIRAttr(UInt(attrindx), walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions, descriptor,
                                                        regime, walkparams);
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecureAArch64.S1ApplyTablePermsP
        regime, walkparams);

then
    // In Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk
    nextstate.nG = '1';
else
    nextstate.permissions = currentstate.permissions;
    nextstate.nG = descriptor<11>;

    nextstate.guardedpage = descriptor<50>;

return nextstate;

```

Library pseudocode for aarch64/translation/  
vmsa\_walk/AArch64.S1WalkAArch64.S1NextWalkStateTable

```

// AArch64.S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor
// AArch64.S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level

(FaultRecord, AddressDescriptor, TTWState, bits(64))TTWState AArch64.S1Walk(AArch64.S1NextWalkStateTable,
    S1TTWParams walkparams, bits(64) va, Regime regime, SecurityStateS1TTWParams ss, walkparams,
    bits(64) descriptor)
    AccType acctype, boolean iswrite_in, boolean ispriv)
    FaultRecord fault = fault_in;
    boolean iswrite = iswrite_in;
    if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
        fault.statuscode = Fault_Translation;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    walkstate = nextstate; AArch64.S1InitialTTWState(walkparams, va, regime, ss);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
        fault.statuscode = Fault_AddressSize;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = va;
    if !AArch64.S1DCacheEnabled(regime) then
        walkaddress.memattr = NormalNCMemAttr();
        walkaddress.memattr.xs = walkstate.memattr.xs;
    else
        walkaddress.memattr = walkstate.memattr;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattr.shareability = walkstate.memattr.shareability;
    else
        walkaddress.memattr.shareability = EffectiveShareability(walkaddress.memattr);

    DescriptorType descctype;
    repeat
        fault.level = walkstate.level;
        FullAddress descaddress = tablebase;

        tablebase.address = AArch64.TTEntryAddressAArch64.NextTableBase(walkstate.level, walkparams.tgx,
            walkparams.txsz, va,
            walkstate.baseaddress);

        walkaddress.paddress = descaddress;

        if regime == (descriptor, walkparams.ds, walkparams.tgx);
        if currentstate.baseaddress.paspace == Regime_EL10PAS_Secure && then
            // Determine PA space of the next table from NSTable bit
            tablebase.paspace = if descriptor<63> == '0' then EL2EnabledPAS_Secure() then
                slaarch64 = TRUE;
                s2fslwalk = TRUE;
                aligned = TRUE;
                iswrite = FALSE;
                (s2fault, s2walkaddress) = else AArch64.S2TranslatePAS_NonSecure(fault, walkaddress, slaarch64,
                    s2fslwalk,
                else
                    // Otherwise bit 63 is RES0 and there is no NSTable bit
                    tablebase.paspace = currentstate.baseaddress.paspace;

```

```

nextstate.istable      = TRUE;
nextstate.nG           = currentstate.nG;
nextstate.level        = currentstate.level + 1;
nextstate.baseaddress  = tablebase;
nextstate.memattrs     = currentstate.memattrs;

if walkparams.hpd == '0' then
    nextstate.permissions = AccType\_TTWAArch64.S1ApplyTablePerms, aligned,
                                iswrite, ispriv);

    if s2fault.statuscode != Fault\_None then
        return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
                                            walkstate.level);

    case desctype of
        when DescriptorType\_Table
            walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams,
                                                    descriptor);

            // Detect Address Size Fault by table descriptor
            if AArch64.OA0OutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
                fault.statuscode = Fault\_AddressSize;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            when DescriptorType\_Page, DescriptorType\_Block
                walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss,
                                                        walkparams, descriptor);

            when DescriptorType\_Invalid
                fault.statuscode = Fault\_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            otherwise
                Unreachable();

    until desctype IN {DescriptorType\_Page, DescriptorType\_Block};

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
        fault.statuscode = Fault\_Translation;
    elsif desctype == DescriptorType\_Block && AArch64.BlocknTFaults(descriptor) then
        fault.statuscode = Fault\_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OA0OutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
        fault.statuscode = Fault\_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
        !(acctype IN {AccType\_DC, AccType\_IC} &&
        !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault\_AccessFlag;
    (currentstate.permissions, descriptor,
                                regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;

    return (fault, walkaddress, walkstate, descriptor); return nextstate;

```

Library pseudocode for aarch64/translation/  
vmsa\_walk/AArch64.S2InitialTTWStateAArch64.S1Walk

```

// AArch64.S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2
// AArch64.S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

TTWState(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2InitialTTWState(AArch64.S1Walk(FaultRecord,
    S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss, S2TTWParamsAccType walkparamsAccType,
    FaultRecord fault = fault_in;
    boolean iswrite = iswrite_in;
    if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
        fault.statuscode = Fault_Translation;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState walkstate; UNKNOWN, bits(64) UNKNOWN);

    walkstate =
        AArch64.S1InitialTTWState(walkparams, va, regime, ss);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
        fault.statuscode = Fault_AddressSize;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = va;
    if !AArch64.S1DCacheEnabled(regime) then
        walkaddress.memattr = NormalNCMemAttr();
        walkaddress.memattr.xs = walkstate.memattr.xs;
    else
        walkaddress.memattr = walkstate.memattr;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattr.shareability = walkstate.memattr.shareability;
    else
        walkaddress.memattr.shareability = EffectiveShareability(walkaddress.memattr);

    DescriptorType descType;
    repeat
        fault.level = walkstate.level;
        FullAddress tablebase;

        ttbr = VTTBR_EL2;
        startlevel = descaddress = AArch64.S2StartLevelAArch64.TTEntryAddress(walkparams);
        tablebase.paspace = (walkstate.level, walkparams.tgx,
            walkparams.txsz, va,
            walkstate.baseaddress);

        walkaddress.paddress = descaddress;

        if regime == PAS_NonSecureRegime_EL10;
        tablebase.address = && AArch64.TTBaseAddressEL2Enabled(ttbr, walkparams.txsz, walkparams.ps, walkparams.tgx, startlevel);

        walkstate.baseaddress = tablebase;
        walkstate.level = startlevel;
        walkstate.istable = TRUE;
        walkstate.memattr = () then
            slaarch64 = TRUE;
            s2fs1walk = TRUE;
            aligned = TRUE;
            iswrite = FALSE;
            (s2fault, s2walkaddress) = (fault, walkaddress, slaarch64, ss,

```

```

s2fslwalk, AccType_TTW, aligned,
iswrite, ispriv);

if s2fault.statuscode != Fault_None then
    return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    (fault, descriptor) = FetchDescriptor(walkparams.tee, s2walkaddress, fault);
else
    (fault, descriptor) = FetchDescriptor(walkparams.tee, walkaddress, fault);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
                                         walkstate.level);

case desctype of
    when DescriptorType_Table
        walkstate = AArch64.SiNextWalkStateTable(walkstate, regime, walkparams,
                                                  descriptor);

        // Detect Address Size Fault by table descriptor
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
            fault.statuscode = Fault_AddressSize;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    when DescriptorType_Page, DescriptorType_Block
        walkstate = AArch64.SiNextWalkStateLast(walkstate, regime, ss,
                                                  walkparams, descriptor);

    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    otherwise
        Unreachable();

until desctype IN {DescriptorType_Page, DescriptorType_Block};

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txs, walkparams.tgx, walkstate.level)) then
    fault.statuscode = Fault_Translation;
elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
    !(acctype IN {AccType_DC, AccType_IC} &&
    !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
    fault.statuscode = Fault_AccessFlagWalkMemAttrsAArch64.S2Translate(walkparams.sh, walkparams.irgn);
;

return walkstate; return (fault, walkaddress, walkstate, descriptor);

```

## Library pseudocode for aarch64/translation/

### vmsa\_walk/AArch64.S2NextWalkStateLastAArch64.S2InitialTTWState

```
// AArch64.S2NextWalkStateLast()
// =====
// Decode stage 2 page or block descriptor as output to this stage of translation
// AArch64.S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2NextWalkStateLast(AArch64.S2InitialTTWState(TTWState currentstate, SecurityState ss,
                                                                    S2TTWParams walkparams, walkparams, AddressDescriptor ipa,
                                                                    bits(64) descriptor)
    TTWState    nextstate; walkstate;
    FullAddress baseaddress;
tablebase;

    if ss == ttbr = VTTBR_EL2;
    startlevel = SS_SecureAArch64.S2StartLevel then
        baseaddress.paspace = (walkparams);
    tablebase.paspace = AArch64.SS20OutputPASpace(walkparams, ipa.paddress.paspace);
    else
        baseaddress.paspace = PAS_NonSecure;

    if currentstate.level == tablebase.address = FINAL_LEVELAArch64.TTBaseAddress then
        baseaddress.address = (ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = AArch64.PageBaseWalkMemAttrs(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx,
                                                    currentstate.level);

    nextstate.istable = FALSE;
    nextstate.level = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
        if s2_attr<1:0> == '10' then // Force writeback
            nextstate.memattrs.xs = '0';
        else
            nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    else
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh);
        nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
        nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);
(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return nextstate; return walkstate;
```

## Library pseudocode for aarch64/translation/

### vmsa\_walk/AArch64.S2NextWalkStateTableAArch64.S2NextWalkStateLast

```
// AArch64.S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level
// AArch64.S2NextWalkStateLast()
// =====
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateTable(AArch64.S2NextWalkStateLast(TTWState currentstate, SecurityState ss,
                                                                    S2TTWParams walkparams,
                                                                    bits(64) descriptor)walkparams,
    AddressDescriptor ipa,
    bits(64) descriptor)
    TTWState nextstate;
    FullAddress tablebase;
    baseaddress;

    tablebase.address = if ss == then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    else
        baseaddress.paspace = PAS_NonSecure;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx,
                                                currentstate.level);

    nextstate.istable = FALSE;
    nextstate.level = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
        if s2_attr<1:0> == '10' then // Force writeback
            nextstate.memattrs.xs = '0';
        else
            nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    else
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh);
        nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    nextstate.contiguous = AArch64.ContiguousBitAArch64.NextTableBaseSS_Secure(descriptor, walkparams.ds,
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable = TRUE;
    nextstate.level = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs = currentstate.memattrs;
    (walkparams.tgx, currentstate.level, descriptor);

    return nextstate;
```

Library pseudocode for aarch64/translation/  
vmsa\_walk/AArch64.S2WalkAArch64.S2NextWalkStateTable

```

// AArch64.S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor
// AArch64.S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level

(FaultRecord, AddressDescriptor, TTWState, bits(64))TTWState AArch64.S2Walk(AArch64.S2NextWalkStateTable,
    FaultRecordTTWState fault_in, currentstate, AddressDescriptor ipa, S2TTWParams walkparams, walkpara
        bits(64) descriptor) SecurityState ss,
    AccType acctype, boolean iswrite, boolean slaarch64)

    FaultRecord fault = fault_in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate;
    if ss == nextstate; SS_Secure then
        walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64.S2InitialTTWState(ss, walkparams);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;
    if HCR_EL2.CD == '1' then
        walkaddress.memattr = NormalNCMemAttr();
        walkaddress.memattr.xs = walkstate.memattr.xs;
    else
        walkaddress.memattr = walkstate.memattr;

    walkaddress.memattr.shareability = EffectiveShareability(walkaddress.memattr);

    DescriptorType descctype;
    repeat
        fault.level = walkstate.level;

        FullAddress descaddress;
        if walkstate.level == tablebase;

            tablebase.address = AArch64.S2StartLevelAArch64.NextTableBase(walkparams) then
                // Initial lookup might index into concatenated tables
                descaddress = AArch64.S2SLTTEnterAddress(walkparams, ipa.paddress.address,
                    walkstate.baseaddress);
            else
                ipa_64 = ZeroExtend(ipa.paddress.address, 64);
                descaddress = AArch64.TTEnterAddress(walkstate.level, walkparams.tgx, walkparams.txsz,
                    ipa_64, walkstate.baseaddress);

        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        descctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
            walkstate.level);

        case descctype of
            when DescriptorType_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);

        // Detect Address Size Fault by table descriptor

```

```

        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
            fault.statuscode = Fault_AddressSize;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        when DescriptorType_Page, DescriptorType_Block
            walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,
                                                    descriptor);

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        otherwise
            Unreachable();

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
        !(acctype IN {AccType_DC, AccType_IC} &&
        !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;
(descriptor, walkparams.ds, walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    return (fault, walkaddress, walkstate, descriptor);
    nextstate.istable = TRUE;
    nextstate.level = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs = currentstate.memattrs;

    return nextstate;

```

Library pseudocode for aarch64/translation/  
vmsa\_walk/AArch64.SS2InitialTTWStateAArch64.S2Walk

```

// AArch64.SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage 2
// AArch64.S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

TTWState(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.SS2InitialTTWState(AArch64.S2Walk(E
    AccType acctype, boolean iswrite, boolean slaarch64)

    FaultRecord fault = fault_in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate; walkstate;
    if ss ==
        FullAddressSS_Secure tablebase;

    bits(64) ttbr;
    if ipaspace == then
        walkstate = PAS_SecureAArch64.SS2InitialTTWState then
            ttbr = VSTTBR_EL2;
(walkparams, ipa.paddress.paspace);
    else
        ttbr = VTTBR_EL2;

    if ipaspace == walkstate = PAS_SecureAArch64.S2InitialTTWState then
        if walkparams.sw == '0' then
            tablebase.paspace = (ss, walkparams);

    // Detect Address Size Fault by TTB
    if PAS_SecureAArch64.OAOutOfRange;
    else
        tablebase.paspace = (walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
            fault.statuscode = PAS_NonSecureFault_AddressSize;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = fault.level = 0;
        return (fault, PAS_SecureAddressDescriptor;
    else
        tablebase.paspace = UNKNOWN, PAS_NonSecureTTWState;
UNKNOWN, bits(64) UNKNOWN);

    startlevel = bits(64) descriptor; AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;
    if HCR_EL2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType descctype;
    repeat
        fault.level = walkstate.level;

        FullAddress descaddress;
        if walkstate.level == AArch64.S2StartLevel(walkparams);
        tablebase.address = (walkparams) then
            // Initial lookup might index into concatenated tables
            descaddress = AArch64.TTBaseAddressAArch64.S2SLTTEnterAddress(ttbr, walkparams.txsz, walkpara
                walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = (walkparams, ipa.paddress.address,
        walkstate.baseaddress);

```

```

else
    ipa_64 = (ipa.paddress.address, 64);
    descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx, walkparams.txsz,
                                         ipa_64, walkstate.baseaddress);

    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
                                             walkstate.level);

    case desctype of
        when DescriptorType_Table
            walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);

            // Detect Address Size Fault by table descriptor
            if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
                fault.statuscode = Fault_AddressSize;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        when DescriptorType_Page, DescriptorType_Block
            walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,
                                                    descriptor);

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        otherwise
            Unreachable();

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif desctype == DescriptorType_Block && AArch64.BlocknIFaults(descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
           !(acctype IN {AccType_DC, AccType_IC}) &&
           !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlagWalkMemAttrsZeroExtend(walkparams.sh, walkparams.irgn, walkpa
;

return walkstate;      return (fault, walkaddress, walkstate, descriptor);

```

## Library pseudocode for aarch64/translation/ vmsa\_walk/AArch64.SS2OutputPASpaceAArch64.SS2InitialTTWState

```
// AArch64.SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation
// AArch64.SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage 2

PASpace TTWState AArch64.SS2OutputPASpace(AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace,
    if ipaspace == ipaspace) TTWState walkstate;
    FullAddress tablebase;

    bits(64) ttbr;
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return ttbr = VSTTBR_EL2;
        else
            ttbr = VTTBR_EL2;

    if ipaspace == PAS_Secure;
    else
        return then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return if walkparams.nsw == '0' then
                tablebase.paspace = PAS_Secure;
            else
                return tablebase.paspace = PAS_NonSecure;

    startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
        walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

## Library pseudocode for aarch64/

translation/vmsa\_walkparamsvmsa\_walk/AArch64.BBMSupportLevelAArch64.SS2OutputPASpace

```
// AArch64.BBMSupportLevel()
// =====
// Returns the level of FEAT_BBM supported
// AArch64.SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation

integer PASpace AArch64.BlockBBMSupportLevel()
    if !AArch64.SS2OutputPASpace( walkparams, PASpace ipaspace)
        if ipaspace == PAS_Secure then
            if walkparams.<sw,sa> == '00' then
                return PAS_Secure;
            else
                return PAS_NonSecure;
        else
            if walkparams.<sw,sa,ns,nsa> == '0000' then
                return PAS_Secure;
            else
                return PAS_NonSecureHaveBlockBBMS2TTWParams() then
                return integer UNKNOWN;
            else
                return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

## Library pseudocode for aarch64/translation/

vmsa\_walkparams/AArch64.DecodeTG0AArch64.BBMSupportLevel

```
// AArch64.DecodeTG0()
// =====
// Decode granule size configuration bits TG0
// AArch64.BBMSupportLevel()
// =====
// Returns the level of FEAT_BBM supported

integer AArch64.DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG0 encoding granule size";

    case tg0 of
        when '00' return AArch64.BlockBBMSupportLevel();
    if ! TGx_4KBHaveBlockBBM;
        when '01' return TGx_64KB;
        when '10' return TGx_16KB;() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

## Library pseudocode for aarch64/translation/

### vmsa\_walkparams/AArch64.DecodeTG1AArch64.DecodeTG0

```
// AArch64.DecodeTG1()
// AArch64.DecodeTG0()
// =====
// Decode granule size configuration bits TG1
// Decode granule size configuration bits TG0

TGx AArch64.DecodeTG1(bits(2) tg1_in)
  bits(2) tg1 = tg1_in;
  if tg1 == '00' then
    tg1 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG1 encoding granule size";
AArch64.DecodeTG0(bits(2) tg0_in)
  bits(2) tg0 = tg0_in;
  if tg0 == '11' then
    tg0 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG0 encoding granule size";

  case tg1 of
    when '10' return case tg0 of
    when '00' return TGx_4KB;
    when '11' return when '01' return TGx_64KB;
    when '01' return when '10' return TGx_16KB;
```

## Library pseudocode for aarch64/translation/

### vmsa\_walkparams/AArch64.GetS1TTWParamsAArch64.DecodeTG1

```
// AArch64.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.
// AArch64.DecodeTG1()
// =====
// Decode granule size configuration bits TG1

S1TTWParamsTGx AArch64.GetS1TTWParams(AArch64.DecodeTG1(bits(2) tg1_in)
  bits(2) tg1 = tg1_in;
  if tg1 == '00' then
    tg1 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG1 encoding granule size";

  case tg1 of
    when '10' return RegimeTGx_4KB regime, bits(64) va);
    when '11' return
      S1TTWParamsTGx_64KB walkparams;

  varange =;
    when '01' return AArch64.GetVARangeTGx_16KB(va);

  case regime of
    when Regime_EL3 walkparams = AArch64.S1TTWParamsEL3();
    when Regime_EL2 walkparams = AArch64.S1TTWParamsEL2();
    when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(varange);
    when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

  maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
  mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
  if UInt(walkparams.txsz) > maxtxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
      walkparams.txsz = maxtxsz<5:0>;
  elsif !Have52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
      walkparams.txsz = mintxsz<5:0>;

  return walkparams;
```

## Library pseudocode for aarch64/translation/

### vmsa\_walkparams/AArch64.GetS2TTWParamsAArch64.GetS1TTWParams

```
// AArch64.GetS2TTWParams()
// AArch64.GetS1TTWParams()
// =====
// Gather walk parameters for stage 2 translation
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.

S2TTWParams S1TTWParams AArch64.GetS2TTWParams(AArch64.GetS1TTWParams(SecurityStateRegime ss, regime, bits-
varange =
S2TTWParams AArch64.GetVARange walkparams;
(va);

if ss == case regime of
when SS_NonSecureRegime_EL3 then
walkparams = walkparams = AArch64.NSS2TTWParams AArch64.S1TTWParamsEL3(slaarch64);
elseif();
when HaveSecureEL2ExtRegime_EL2() && ss == walkparams = SS_Secure AArch64.S1TTWParamsEL2 then
walkparams = ();
when AArch64.SS2TTWParamsRegime_EL20(ipaspace, slaarch64);
else walkparams =
Unreachable AArch64.S1TTWParamsEL20();

maxtxsz = (varange);
when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
mintxsz = AArch64.S2MinTxSZ AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
(walkparams.ds, walkparams.tgx);
if UInt(walkparams.txsz) > maxtxsz then
if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
walkparams.txsz = maxtxsz<5:0>;
elseif !Have52BitPAExtHave52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
walkparams.txsz = mintxsz<5:0>;

return walkparams;
```

**Library pseudocode for aarch64/translation/  
vmsa\_walkparams/AArch64.GetVARangeAArch64.GetS2TTWParams**

```
// AArch64.GetVARange()
// =====
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.
// AArch64.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

VARangeS2TTWParams AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        returnAArch64.GetS2TTWParams( VARange_LOWERSecurityState;
    else
        returnss, ipaspace, boolean slaarch64)
        S2TTWParams walkparams;

    if ss == SS_NonSecure then
        walkparams = AArch64.NSS2TTWParams(slaarch64);
    elsif HaveSecureEL2Ext() && ss == SS_Secure then
        walkparams = AArch64.SS2TTWParams(ipaspace, slaarch64);
    else
        Unreachable();

    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
    elsif !HaveS2BitPAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "FaultVARange_UPPERPASpace; on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;

    return walkparams;
```

**Library pseudocode for aarch64/translation/  
vmsa\_walkparams/AArch64.MaxTxSZAArch64.GetVARange**

```
// AArch64.MaxTxSZ()
// =====
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation
// AArch64.GetVARange()
// =====
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

integerVARange AArch64.MaxTxSZ(AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        returnTGxVARange_LOWER tgx)
    if;
    else
        return HaveSmallTranslationTableExtVARange_UPPER() && !UsingAArch32() then
            case tgx of
                when TGx_4KB return 48;
                when TGx_16KB return 48;
                when TGx_64KB return 47;
            return 39;;
```

**Library pseudocode for aarch64/translation/  
vmsa\_walkparams/AArch64.NSS2TTWParamsAArch64.MaxTxSZ**

```
// AArch64.NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation
// AArch64.MaxTxSZ()
// =====
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

S2TTWParams integer AArch64.NSS2TTWParams(boolean slaarch64) AArch64.MaxTxSZ(
    S2TTWParams TGx walkparams;

    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx = tgx;
    if AArch64.DecodeTG0HaveSmallTranslationTableExt(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.sl0 = VTCR_EL2.SL0;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    walkparams.sh = VTCR_EL2.SH0;
    walkparams.ee = SCTL_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
    walkparams.fwb = if() && ! HaveStage2MemAttrControlUsingAArch32() then HCR_EL2.FWB else '0';
    walkparams.ha = if() then
        case tgx of
            when HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, return 48;
        when TGx_16KB} && return 48;
        when Have52BitIPAAAndPASpaceExtTGx_64KB() then
            walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';

    return walkparams; return 47;
    return 39;
```

## Library pseudocode for aarch64/translation/ vmsa\_walkparams/AArch64.PAMaxAArch64.NSS2TTWParams

```
// AArch64.PAMax()
// =====
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this processor
// AArch64.NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation

integer S2TTWParams AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size"; AArch64.NSS2TTWParams(boolean - s

    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.sl0 = VTCR_EL2.SL0;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    walkparams.sh = VTCR_EL2.SH0;
    walkparams.ee = SCTLRL_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';

    return walkparams;
```

## Library pseudocode for aarch64/translation/ vmsa\_walkparams/AArch64.S1DCacheEnabledAArch64.PAMax

```
// AArch64.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses
// AArch64.PAMax()
// =====
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this processor

booleaninteger AArch64.S1DCacheEnabled(AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size"; Regime regime)
    case regime of
        when Regime_EL3 return SCTLRL_EL3.C == '1';
        when Regime_EL2 return SCTLRL_EL2.C == '1';
        when Regime_EL20 return SCTLRL_EL2.C == '1';
        when Regime_EL10 return SCTLRL_EL1.C == '1';
```

## Library pseudocode for aarch64/translation/

### vmsa\_walkparams/AArch64.S1EPDAArch64.S1DCacheEnabled

```
// AArch64.S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the VA range
// AArch64.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

bitboolean AArch64.S1EPD(AArch64.S1DCacheEnabled(Regime regime, bits(64) va)
    assert(regime);
    case regime of
        when HasUnprivilegedRegime_EL3(regime);
            varange = return SCTL_EL3.C == '1';
        when AArch64.GetVARangeRegime_EL2(va);

    case regime of
return SCTL_EL2.C == '1';
    when Regime_EL20 return if varange == return SCTL_EL2.C == '1';
    when VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
    when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1; return S
```

## Library pseudocode for aarch64/translation/

### vmsa\_walkparams/AArch64.S1EnabledAArch64.S1EPD

```
// AArch64.S1Enabled()
// =====
// Determine if stage 1 for the acting translation regime is enabled
// AArch64.S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the VA range

booleanbit AArch64.S1Enabled(AArch64.S1EPD(Regime regime)
    case regime of
        when regime, bits(64) va)
            assert Regime_EL3HasUnprivileged return SCTL_EL3.M == '1';
        when (regime);
            varange = Regime_EL2AArch64.GetVARange return SCTL_EL2.M == '1';
(va);

    case regime of
        when Regime_EL20 return SCTL_EL2.M == '1';
        when return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return (!return if varange == EL2EnabledVARange_LOWER() || HCR_EL2.<DC,TGE> == '0
```

## Library pseudocode for aarch64/translation/

### vmsa\_walkparams/AArch64.S1ICacheEnabledAArch64.S1Enabled

```
// AArch64.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches
// AArch64.S1Enabled()
// =====
// Determine if stage 1 for the acting translation regime is enabled

boolean AArch64.S1ICacheEnabled(AArch64.S1Enabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTL_EL3.I == '1';
return SCTL_EL3.M == '1';
        when Regime_EL2 return SCTL_EL2.I == '1';
return SCTL_EL2.M == '1';
        when Regime_EL20 return SCTL_EL2.I == '1';
return SCTL_EL2.M == '1';
        when Regime_EL10 return SCTL_EL1.I == '1'; return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') &&
```

## Library pseudocode for aarch64/translation/

vmsa\_walkparams/**AArch64.S1MinTxSZ****AArch64.S1ICacheEnabled**

```
// AArch64.S1MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1
// AArch64.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

integer boolean AArch64.S1MinTxSZ(bit ds, AArch64.S1ICacheEnabled( TGxRegime tgx)
    if (regime)
        case regime of
            when Have52BitVAExtRegime_EL3() && tgx == return SCTLRL_EL3.I == '1';
            when return SCTLRL_EL2.I == '1';
            when Regime_EL20 return SCTLRL_EL2.I == '1';
            when Regime_EL10TGx_64KBRegime_EL2) || ds == '1' then
                return 12;

    return 16;return SCTLRL_EL1.I == '1';
```

## Library pseudocode for aarch64/translation/

vmsa\_walkparams/**AArch64.S1TTBR****AArch64.S1MinTxSZ**

```
// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime
// AArch64.S1MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

bits(64)integer AArch64.S1TTBR(AArch64.S1MinTxSZ(bit ds,RegimeTGx regime, bits(64) va)
    varange =tgx);
    if ( AArch64.GetVARangeHave52BitVAExt(va);

        case regime of
            when() && tgx == Regime_EL3TGx_64KB return TTBR0_EL3;
            when Regime_EL2 return TTBR0_EL2;
            when Regime_EL20 return if varange == VARange_LOWER then TTBR0_EL2 else TTBR1_EL2;
            when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1; ) || ds == '1'
                return 12;

    return 16;
```

Library pseudocode for aarch64/translation/  
vmsa\_walkparams/AArch64.S1TTWParamsEL10AArch64.S1TTBR

```

// AArch64.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)
// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime

S1TTWParamsbits(64) AArch64.S1TTWParamsEL10(AArch64.S1TTBR(VARangeRegime varange) regime, bits(64) va)
    varange =
        S1TTWParamsAArch64.GetVARange walkparams;
(va);

    if varange == case regime of
        when VARange_LOWERRegime_EL3 then
            walkparams.tgx = return TTBR0_EL3;
        when AArch64.DecodeTG0Regime_EL2(TCR_EL1.TG0);
            walkparams.txsz = TCR_EL1.T0SZ;
            walkparams.irgn = TCR_EL1.IRGN0;
            walkparams.orgn = TCR_EL1.ORGNO;
            walkparams.sh = TCR_EL1.SH0;
            walkparams.tbi = TCR_EL1.TBI0;

            walkparams.tbid = if return TTBR0_EL2;
            when HavePACExtRegime_EL20() then TCR_EL1.TBID0 else '0';
            walkparams.e0pd = if return if varange == HaveE0PDExtVARange_LOWER() then TCR_EL1.E0PD0 else '0';
            walkparams.hpd = if then TTBR0_EL2 else TTBR1_EL2;
            when AArch64.HaveHPDExtRegime_EL10() then TCR_EL1.HPD0 else '0';
        else
            walkparams.tgx = return if varange == AArch64.DecodeTG1VARange_LOWER(TCR_EL1.TG1);
            walkparams.txsz = TCR_EL1.T1SZ;
            walkparams.irgn = TCR_EL1.IRGN1;
            walkparams.orgn = TCR_EL1.ORGNO;
            walkparams.sh = TCR_EL1.SH1;
            walkparams.tbi = TCR_EL1.TBI1;

            walkparams.tbid = if HavePACExt() then TCR_EL1.TBID1 else '0';
            walkparams.e0pd = if HaveE0PDExt() then TCR_EL1.E0PD1 else '0';
            walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL1.HPD1 else '0';

            walkparams.mair = MAIR_EL1;
            walkparams.wxn = SCTL_EL1.WXN;
            walkparams.ps = TCR_EL1.IPS;
            walkparams.ee = SCTL_EL1.EE;
            walkparams.sif = SCR_EL3.SIF;

            if EL2Enabled() then
                walkparams.dc = HCR_EL2.DC;
                walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

            if HaveTrapLoadStoreMultipleDeviceExt() then
                walkparams.ntlsm = SCTL_EL1.nTlsm;
            else
                walkparams.ntlsm = '1';

            if EL2Enabled() then
                if HCR_EL2.<NV,NV1> == '01' then
                    case ConstrainUnpredictable(Unpredictable_NVNV1) of
                        when Constraint_NVNV1_00 walkparams.nv1 = '0';
                        when Constraint_NVNV1_01 walkparams.nv1 = '1';
                        when Constraint_NVNV1_11 walkparams.nv1 = '1';
                    else
                        walkparams.nv1 = HCR_EL2.NV1;
                else
                    walkparams.nv1 = '0';

            walkparams.epan = if HavePAN3Ext() then SCTL_EL1.EPAN else '0';
            walkparams.cmow = if HaveFeatCMOW() then SCTL_EL1.CMOW else '0';
            walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
            walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';

```

```
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
    walkparams.ds = TCR_EL1.DS;
else
    walkparams.ds = '0';

return walkparams;then TTBR0_EL1 else TTBR1_EL1;
```

Library pseudocode for aarch64/translation/  
vmsa\_walkparams/AArch64.S1TTWParamsEL2AArch64.S1TTWParamsEL10

```

// AArch64.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime
// AArch64.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL2() AArch64.S1TTWParamsEL10(
    VARange varange)
    S1TTWParams walkparams;

    walkparams.tgx = if varange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;
    (TCR_EL1.TG0);
    walkparams.txsz = TCR_EL1.T0SZ;
    walkparams.irgn = TCR_EL1.IRGN0;
    walkparams.orgn = TCR_EL1.ORGNO;
    walkparams.sh = TCR_EL1.SH0;
    walkparams.tbi = TCR_EL1.TBI0;

    walkparams.tbid = if walkparams.tbid = if HavePACEExt() then TCR_EL2.TBID else '0';
    walkparams.hpd = if() then TCR_EL1.TBID0 else '0';
    walkparams.e0pd = if HaveE0PDEExt() then TCR_EL1.E0PD0 else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz = TCR_EL1.T1SZ;
        walkparams.irgn = TCR_EL1.IRGN1;
        walkparams.orgn = TCR_EL1.ORGNO;
        walkparams.sh = TCR_EL1.SH1;
        walkparams.tbi = TCR_EL1.TBI1;

        walkparams.tbid = if HavePACEExt() then TCR_EL1.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL1.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD1 else '0';

        walkparams.mair = MAIR_EL1;
        walkparams.wxn = SCTLR_EL1.WXN;
        walkparams.ps = TCR_EL1.IPS;
        walkparams.ee = SCTLR_EL1.EE;
        walkparams.sif = SCR_EL3.SIF;

    if EL2Enabled() then
        walkparams.dc = HCR_EL2.DC;
        walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTLR_EL1.nTLSMD;
    else
        walkparams.ntlsmid = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';
            else
                walkparams.nv1 = HCR_EL2.NV1;

```

```

else
    walkparams.nv1 = '0';

    walkparams.epan = if HavePAN3Ext() then SCTL_EL1.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW() then TCR_EL2.HPD else '0';
    () then SCTL_EL1.CMOW else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    () then TCR_EL1.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    () then TCR_EL1.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '0';

    return walkparams;

```

## Library pseudocode for aarch64/translation/

vmsa\_walkparams/~~AArch64.S1TTWParamsEL20~~~~AArch64.S1TTWParamsEL2~~

```
// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime
// AArch64.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL20(AArch64.S1TTWParamsEL2() VARange varange)
    S1TTWParams walkparams;

    if varange == walkparams.tgx == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(TCR_EL2.TG0);
        walkparams.txsz = TCR_EL2.T0SZ;
        walkparams.irgn = TCR_EL2.IRGN0;
        walkparams.orgn = TCR_EL2.ORGNO;
        walkparams.sh = TCR_EL2.SH0;
        walkparams.tbi = TCR_EL2.TBI0;
        walkparams.txsz = TCR_EL2.T0SZ;
        walkparams.ps = TCR_EL2.PS;
        walkparams.irgn = TCR_EL2.IRGN0;
        walkparams.orgn = TCR_EL2.ORGNO;
        walkparams.sh = TCR_EL2.SH0;
        walkparams.tbi = TCR_EL2.TBI;
        walkparams.mair = MAIR_EL2;
        walkparams.wxn = SCTLR_EL2.WXN;
        walkparams.ee = SCTLR_EL2.EE;
        walkparams.sif = SCR_EL3.SIF;

        walkparams.tbid = if walkparams.tbid = if HavePACExt() then TCR_EL2.TBID0 else '0';
        walkparams.e0pd = if() then TCR_EL2.TBID else '0';
        walkparams.hpd = if HaveE0PDEExt() then TCR_EL2.E0PD0 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz = TCR_EL2.T1SZ;
        walkparams.irgn = TCR_EL2.IRGN1;
        walkparams.orgn = TCR_EL2.ORGNO;
        walkparams.sh = TCR_EL2.SH1;
        walkparams.tbi = TCR_EL2.TBI1;

        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD1 else '0';

        walkparams.mair = MAIR_EL2;
        walkparams.wxn = SCTLR_EL2.WXN;
        walkparams.ps = TCR_EL2.IPS;
        walkparams.ee = SCTLR_EL2.EE;
        walkparams.sif = SCR_EL3.SIF;

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTLR_EL2.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    walkparams.epan = if HavePAN3Ext() then SCTLR_EL2.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW() then SCTLR_EL2.CMOW else '0';
    () then TCR_EL2.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/  
vmsa\_walkparams/AArch64.S1TTWParamsEL3AArch64.S1TTWParamsEL20

```

// AArch64.S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime
// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL3()AArch64.S1TTWParamsEL20(
    VARange varange)
    S1TTWParams walkparams;

    walkparams.tgx = if varange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGNO;
    walkparams.sh = TCR_EL3.SH0;
    walkparams.tbi = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    walkparams.wxn = SCTLR_EL3.WXN;
    walkparams.ee = SCTLR_EL3.EE;
    walkparams.sif = SCR_EL3.SIF;
    (TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI0;

    walkparams.tbid = if walkparams.tbid = if HavePACEExt() then TCR_EL3.TBID else '0';
    walkparams.hpd = if() then TCR_EL2.TBID0 else '0';
    walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD0 else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz = TCR_EL2.T1SZ;
        walkparams.irgn = TCR_EL2.IRGN1;
        walkparams.orgn = TCR_EL2.ORGNO;
        walkparams.sh = TCR_EL2.SH1;
        walkparams.tbi = TCR_EL2.TBI1;

        walkparams.tbid = if HavePACEExt() then TCR_EL2.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD1 else '0';

    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ps = TCR_EL2.IPS;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.epan = if HavePAN3Ext() then SCTLR_EL2.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW() then TCR_EL3.HPD else '0';
    () then SCTLR_EL2.CMOW else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL3.HA else '0';
    () then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL3.HD else '0';
    () then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL3.DS;
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

```

```
return walkparams;
```

## Library pseudocode for aarch64/translation/

vmsa\_walkparams/**AArch64.S2MinTxSZ****AArch64.S1TTWParamsEL3**

```
// AArch64.S2MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2
// AArch64.S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime

integer S1TTWParams AArch64.S2MinTxSZ(bit ds, AArch64.S1TTWParamsEL3() TGxS1TTWParams tgx, boolean slaarch64)
    ips = walkparams;

    walkparams.tgx = AArch64.PAMaxAArch64.DecodeTG0();
(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGNO;
    walkparams.sh = TCR_EL3.SH0;
    walkparams.tbi = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    walkparams.wxn = SCTLRL_EL3.WXN;
    walkparams.ee = SCTLRL_EL3.EE;
    walkparams.sif = SCR_EL3.SIF;

    if walkparams.tbid = if Have52BitPAExtHavePACExt() && tgx !=() then TCR_EL3.TBID else
    walkparams.hpd = if TGx_64KBAArch64.HaveHPDExt && ds == '0' then
        ips =() then TCR_EL3.HPD else '0';
    walkparams.ha = if MinHaveAccessFlagUpdateExt(48,()) then TCR_EL3.HA else '0';
    walkparams.hd = if AArch64.PAMaxHaveDirtyBitModifierExt();

    min_txsz = 64 - ips;
    if !slaarch64 then
        // EL1 is AArch32
        min_txsz =() then TCR_EL3.HD else '0';
    if walkparams.tgx IN { , TGx_16KB } && Have52BitIPAAndPASpaceExtMinTGx_4KB(min_txsz, 24);
() then
    walkparams.ds = TCR_EL3.DS;
    else
    walkparams.ds = '0';

    return min_txsz; return walkparams;
```

## Library pseudocode for aarch64/translation/

vmsa\_walkparams/AArch64.SS2TTWParamsAArch64.S2MinTxSZ

```
// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation
// AArch64.S2MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

S2TTWParams integer AArch64.SS2TTWParams(AArch64.S2MinTxSZ(bit ds, PAspace TGx ipaspace, boolean slaarch64))
    ips =
    S2TTWParams AArch64.PAMax walkparams;
    ();

    if ipaspace == if PAS_SecureHave52BitPAExt then
        walkparams.tgx = () && tgx != AArch64.DecodeTG0TGx_64KB(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        walkparams.sl0 = VSTCR_EL2.SL0;
        if walkparams.tgx == && ds == '0' then
            ips = TGx_4KBMin && (48, Have52BitIPAAAndPASpaceExtAArch64.PAMax()) then
                walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
            else
                walkparams.sl2 = '0';
        elsif ipaspace == ();
        min_txsz = 64 - ips;
        if !slaarch64 then
            // EL1 is AArch32
            min_txsz = PAS_NonSecureMin then
                walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
                walkparams.txsz = VTCR_EL2.T0SZ;
                walkparams.sl0 = VTCR_EL2.SL0;
                if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
                    walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
                else
                    walkparams.sl2 = '0';
            else
                Unreachable();

        walkparams.sw = VSTCR_EL2.SW;
        walkparams.nsw = VTCR_EL2.NSW;
        walkparams.sa = VSTCR_EL2.SA;
        walkparams.nsa = VTCR_EL2.NSA;
        walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
        walkparams.ps = VTCR_EL2.PS;
        walkparams.irgn = VTCR_EL2.IRGN0;
        walkparams.orgn = VTCR_EL2.ORGNO;
        walkparams.sh = VTCR_EL2.SH0;
        walkparams.ee = SCTLR_EL2.EE;

        walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
        walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
        walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
        walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
        if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
            walkparams.ds = VTCR_EL2.DS;
        else
            walkparams.ds = '0';
        walkparams.cmov = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
    (min_txsz, 24);

    return walkparams; return min_txsz;
```

## Library pseudocode for aarch64/translation/ vmsa\_walkparams/AArch64.VAMaxAArch64.SS2TTWParams

```
// AArch64.VAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED maximum number of bits capable of representing
// the virtual address for this processor
// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation

integer SS2TTWParams AArch64.VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size"; AArch64.SS2TTWParams(PASpace - ipa
    SS2TTWParams walkparams;

    if ipaspace == PAS_Secure then
        walkparams.tgx = AArch64.DecodeTG0(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        walkparams.sl0 = VSTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
            walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    elseif ipaspace == PAS_NonSecure then
        walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
        walkparams.txsz = VTCR_EL2.T0SZ;
        walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    else
        Unreachable();

    walkparams.sw = VSTCR_EL2.SW;
    walkparams.nsw = VTCR_EL2.NSW;
    walkparams.sa = VSTCR_EL2.SA;
    walkparams.nsa = VTCR_EL2.NSA;
    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    walkparams.sh = VTCR_EL2.SH0;
    walkparams.ee = SCTL2_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';

    return walkparams;
```

## Library pseudocode for

**sharedAArch64/debugtranslation/ClearStickyErrorsvmsa\_walkparams/ClearStickyErrorsAArch64.VAMa**

```
// ClearStickyErrors()
// =====// AArch64.VAMax()
// =====
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// the virtual address for this processor

integer

ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RXO = '0';           // Clear RX overrun flag

    ifAArch64.VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size"; Halted() then // in
        EDSCR.ITO = '0';       // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';           // Clear cumulative error flag

    return;
```

## Library pseudocode for shared/

**debug/DebugTargetClearStickyErrors/DebugTargetClearStickyErrors**

```
// DebugTarget()
// =====
// Returns the debug exception target Exception level

bits(2)// ClearStickyErrors()
// ===== DebugTarget()
    ss =ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RXO = '0';           // Clear RX overrun flag

    if CurrentSecurityStateHalted();
    return() then             // in Debug state
        EDSCR.ITO = '0';       // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if () && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERODebugTargetFrom
        return;
    EDSCR.ERR = '0';           // Clear cumulative error flag

    return;
```

## Library pseudocode for shared/debug/DebugTarget/DebugTargetFromDebugTarget

```
// DebugTargetFrom()
// =====
// DebugTarget()
// =====
// Returns the debug exception target Exception level

bits(2) DebugTargetFrom(DebugTarget()
    secure = SecurityStateIsSecure from_state)
    boolean route_to_el2;
    if();
    return HaveELDebugTargetFrom(EL2) && (from_state != SS_Secure ||
        (HaveSecureEL2Ext() && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1')))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;

    return target; (secure);
```

## Library pseudocode for shared/debug/DoubleLockStatusDebugTarget/DoubleLockStatusDebugTargetFrom

```
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.
// DebugTargetFrom()
// =====

booleanbits(2) DoubleLockStatus()
    if !DebugTargetFrom(boolean secure)
        boolean route_to_el2;
        ifHaveDoubleLockHaveEL() then
            return FALSE;
        elsif( EL2) && (!secure || (HaveSecureEL2Ext() &&
            (!HaveEL(EL3) || SCR_EL3.EEL2 == '1')))) then
            if ELUsingAArch32(EL1EL2) then
                return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && ! route_to_el2 = (HDCR.TDE == '1');
            else
                route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
        else
            route_to_el2 = FALSE;

        bits(2) target;
        if route_to_el2 then
            target = HaltedEL2();
        else
            return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !;
        elsif(EL3) && !HaveAArch64() && secure then
            target = EL3;
        else
            target = EL1HaltedHaveEL();

    return target;
```

## Library pseudocode for shared/ debug/**OSLockStatusDoubleLockStatus/OSLockStatusDoubleLockStatus**

```
// OSLockStatus()
// =====
// Returns the state of the OS Lock.
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean OSLockStatus()
    return (ifDoubleLockStatus()
        if ! HaveDoubleLock() then
            return FALSE;
        elsif ELUsingAArch32(EL1) then
            return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
        else
            return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted() then DBGOSLSR.OSLK else 0
```

## Library pseudocode for shared/ debug/**SoftwareLockStatusOSLockStatus/ComponentOSLockStatus**

```
enumeration// OSLockStatus()
// =====
// Returns the state of the OS Lock.

boolean Component {OSLockStatus()
    return (if
        Component_PMU, {
        Component_Debug,
        Component_CTI
    };) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

## Library pseudocode for shared/debug/**SoftwareLockStatus/GetAccessComponentComponent**

```
// Returns the accessed component.
Componentenumeration GetAccessComponent();Component {Component_PMU,
    Component_Debug,
    Component_CTI
};
```

## Library pseudocode for shared/debug/ **SoftwareLockStatus/SoftwareLockStatusGetAccessComponent**

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean// Returns the accessed component.
Component SoftwareLockStatus()GetAccessComponent();
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
        otherwise
            Unreachable();
```

## Library pseudocode for shared/

### debug/authenticationSoftwareLockStatus/AccessStateSoftwareLockStatus

```
// Returns the Security state of the access.
SecurityState// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean AccessState();SoftwareLockStatus()Component component = GetAccessComponent();
if !HaveSoftwareLock(component) then
    return FALSE;
case component of
    when Component_Debug
        return EDLSR.SLK == '1';
    when Component_PMU
        return PMLSR.SLK == '1';
    when Component_CTI
        return CTILSR.SLK == '1';
    otherwise
        Unreachable();
```

## Library pseudocode for shared/debug/authentication/AllowExternalDebugAccessAccessState

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean// Returns the Security state of the access.
SecurityState AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    returnAccessState(); AllowExternalDebugAccess(AccessState());

// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

## Library pseudocode for shared/debug/ authentication/~~AllowExternalPMUAccess~~~~AllowExternalDebugAccess~~

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccessAllowExternalDebugAccess(AccessState());

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EPMAD_bit = if _____ EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    ) then SDCR.EDAD else MDCR_EL3.EDAD;
    return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

## Library pseudocode for shared/debug/ authentication/**Debug\_authenticationAllowExternalPMUAccess**

```

signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN; // AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess(AccessState());

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EPMAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation();

```

## Library pseudocode for shared/debug/ authentication/**ExternalInvasiveDebugEnabledDebug\_authentication**

```

// ExternalInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN; ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;

```

**Library pseudocode for shared/debug/  
authentication/ExternalNoninvasiveDebugAllowedExternalInvasiveDebugEnabled**

```
// ExternalNoninvasiveDebugAllowed()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed
// ExternalInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGGEN signal.

boolean ExternalNoninvasiveDebugAllowed()
    if !ExternalInvasiveDebugEnabled()
        return DBGGEN == HIGH; ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(PSTATE.EL);
    if (ELUsingAArch32(EL1) && PSTATE.EL == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;
    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure return ExternalSecureNoninvasiveDebugEnabled();
```

**Library pseudocode for shared/debug/  
authentication/ExternalNoninvasiveDebugEnabledExternalNoninvasiveDebugAllowed**

```
// ExternalNoninvasiveDebugEnabled()
// ExternalNoninvasiveDebugAllowed()
// =====
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGGEN OR NIDEN) signal.
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugEnabled()
    return !ExternalNoninvasiveDebugAllowed()
    if !HaveNoninvasiveDebugAuthExternalNoninvasiveDebugEnabled() || () then return FALSE;
    ss = {PSTATE.EL};
    if (ELUsingAArch32(EL1) && PSTATE.EL == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;
    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure return ExternalSecureNoninvasiveDebugEnabledExternalInvasiveDebugEnabledSecurityStateAtEL(PSTATE.EL);
```

**Library pseudocode for shared/debug/  
authentication/ExternalSecureInvasiveDebugEnabledExternalNoninvasiveDebugEnabled**

```
// ExternalSecureInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGGEN, but this is not recommended.
// ExternalNoninvasiveDebugEnabled()
// =====
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGGEN OR NIDEN) signal.

boolean ExternalSecureInvasiveDebugEnabled()
    if !ExternalNoninvasiveDebugEnabled()
        return !HaveELHaveNoninvasiveDebugAuth(EL3) && !SecureOnlyImplementation() then return FALSE;
    return() || ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;() || NIDEN == HIGH;
```

### Library pseudocode for shared/debug/

authentication/**ExternalSecureNoninvasiveDebugEnabled****ExternalSecureInvasiveDebugEnabled**

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.
// ExternalSecureInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureNoninvasiveDebugEnabled()
ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if return HaveNoninvasiveDebugAuthExternalInvasiveDebugEnabled() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();() && SPIDEN == HIGH;
```

### Library pseudocode for shared/debug/

authentication/**IsAccessSecure****ExternalSecureNoninvasiveDebugEnabled**

```
// Returns TRUE when an access is Secure
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean IsAccessSecure();ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();
```

### Library pseudocode for shared/debug/authentication/**IsCorePowered****IsAccessSecure**

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
// Returns TRUE when an access is Secure
boolean IsCorePowered();IsAccessSecure();
```

## Library pseudocode for shared/ debug/breakpoint/authentication/CheckValidStateMatchIsCorePowered

```
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained
// Unpredictable behaviour, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2))// Returns TRUE if the Core power domain is powered on, FALSE otherwise
boolean CheckValidStateMatch(bits(2) SSC_in, bit HMC_in, bits(2) PxC_in,
                             boolean isbreakpnt)
    boolean reserved = FALSE;
    bits(2) SSC = SSC_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;

    // Values that are not allocated in any architecture version
    if (HMC:SSC:PxC) IN {'01110', '100x0', '10110', '11x10'} then
        reserved = TRUE;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !IsCorePowered(); HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && SSC IN {'01', '10'} && HMC:SSC:PxC != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && HMC:SSC:PxC == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !HaveSecureEL2Ext() && (HMC:SSC:PxC) IN {'01100', '10100', 'x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, SSC, HMC, PxC);
```

## Library pseudocode for shared/debug/ breakpoint/NumBreakpointsImplementedCheckValidStateMatch

```
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented. This is indicated to software by
// DBGDIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained
// Unpredictable behaviour, otherwise returns Constraint_NONE.

integer(Constraint, bits(2), bit, bits(2)) NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints"; CheckValidStateMatch(bits(2) SSC_in, bit
                                                boolean isbreakpnt)

    boolean reserved = FALSE;
    bits(2) SSC = SSC_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;

    // Values that are not allocated in any architecture version
    if (HMC:SSC:PxC) IN {'01110', '100x0', '10110', '11x10'} then
        reserved = TRUE;

    // Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && SSC IN {'01', '10'} && HMC:SSC:PxC != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && HMC:SSC:PxC == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !HaveSecureEL2Ext() && (HMC:SSC:PxC) IN {'01100', '10100', 'x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <HMC, SSC, PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, SSC, HMC, PxC);
```

### Library pseudocode for shared/debug/

#### breakpoint/**NumContextAwareBreakpointsImplemented****NumBreakpointsImplemented**

```
// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented. This is indicated to software by
// DBGDIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented. This is indicated to software by
// DBGDIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

### Library pseudocode for shared/debug/

#### breakpoint/**NumWatchpointsImplemented****NumContextAwareBreakpointsImplemented**

```
// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented. This is indicated to software by
// DBGDIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.
// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented. This is indicated to software by
// DBGDIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

### Library pseudocode for shared/

#### debug/**ctibreakpoint/CTI\_SetEventLevel****NumWatchpointsImplemented**

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented. This is indicated to software by
// DBGDIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.

integer CrossTriggerIn id, signal level);NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

### Library pseudocode for shared/debug/cti/**CTI\_SignalEvent****CTI\_SetEventLevel**

```
// Signal a discrete event on a Cross Trigger input event trigger.// Set a Cross Trigger multi-cycle input event trigger.
CTI_SetEventLevel(
CTI_SignalEvent(CrossTriggerIn id);id, signal level);
```

### Library pseudocode for shared/debug/cti/**CrossTrigger****CTI\_SignalEvent**

```
enumeration// Signal a discrete event on a Cross Trigger input event trigger. CrossTriggerOut {CTI_SignalEvent
    CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
    CrossTriggerOut_TraceExtIn0,   CrossTriggerOut_TraceExtIn1,
    CrossTriggerOut_TraceExtIn2,   CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn {CrossTriggerIn_CrossHalt,      CrossTriggerIn_PMUOverflow,
    CrossTriggerIn_RSVD2,          CrossTriggerIn_RSVD3,
    CrossTriggerIn_TraceExtOut0,    CrossTriggerIn_TraceExtOut1,
    CrossTriggerIn_TraceExtOut2,    CrossTriggerIn_TraceExtOut3};id);
```

## Library pseudocode for shared/debug/~~dccanditrcti~~/CheckForDCCInterruptsCrossTrigger

```
// CheckForDCCInterrupts()
// =====enumeration

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    ifCrossTriggerOut { ELUsingAArch32(CrossTriggerOut_DebugRequest, EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                    (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                    (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(CrossTriggerOut_RestartRequest, CrossTriggerOut_IRQ, CrossTriggerOut
        CrossTriggerOut_TraceExtIn0, CrossTriggerOut_TraceExtIn1,
        CrossTriggerOut_TraceExtIn2, CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn {CrossTriggerIn_CrossHalt, CrossTriggerIn_PMUOverflow,
    CrossTriggerIn_RSVD2, CrossTriggerIn_RSVD3,
    CrossTriggerIn_TraceExtOut0, CrossTriggerIn_TraceExtOut1,
    CrossTriggerIn_TraceExtOut2, InterruptID_COMMIRQ, if commirq then HIGH else

    return; CrossTriggerIn_TraceExtOut3};
```

## Library pseudocode for shared/debug/dccanditr/DBGDTRRX\_EL0CheckForDCCInterrupts

```
// DBGDTRRX_EL0[] (external write)
// =====
// Called on writes to debug register 0x08C.// CheckForDCCInterrupts()
// =====

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value
CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;
    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    if EDSCR.ERR == '1' then return; // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || ( // The value to be driven onto the common COMMIRQ signal.
        boolean commirq;
        if HaltedELUsingAArch32() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
            EDSCR.RX0 = '1'; EDSCR.ERR = '1'; // Overrun condition: ignore write
            return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if( HaltedEL1() && EDSCR.MA == '1' then
        EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
        if ! then
            commirq = ((commrx && DBGDCCINT.RX == '1') ||
                (commtx && DBGDCCINT.TX == '1'));
        else
            commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                (commtx && MDCCINT_EL1.TX == '1'));
        SetInterruptRequestLevel(UsingAArch32InterruptID_COMMIRQ() then
            ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
            // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
            if EDSCR.ERR == '1' then
                EDSCR.RXfull = bit UNKNOWN;
                DBGDTRRX_EL0 = bits(64) UNKNOWN;
            else
                // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
                assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
        return;

// DBGDTRRX_EL0[] (external read)
// =====

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX; if commirq then HIGH else LOW;

return;
```



```

// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

bits(32)// DBGDTRRX_EL0[] (external write)
// =====
// Called on writes to debug register 0x08C. DBGDTRTX_EL0[boolean memory_mapped]
DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;
    return;

    underrun = EDSCR.TXfull == '0' || ( if EDSCR.ERR == '1' then return; // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;
    () && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RX0 = '1'; EDSCR.ERR = '1'; // Overrun condition: ignore write
        return;

    if EDSCR.ERR == '1' then return value; // Error flag set: no side-effects
    EDSCR.RXfull = '1';
    DTRRX = value;

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1'; // Underrun condition: block side-effects
        return value; // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then // See comments in EDITR[] (external write)
        EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(0xB8404401<31:0>); // A64 "LDR W1,[X0],#4"
    else(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteT32ExecuteA64(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.TXfull = bit UNKNOWN;
        DBGDTRTX_EL0 = bits(64) UNKNOWN;
    else
        if !(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"UsingAArch32X() then[1] = bits(64) UNKNOWN;
        else
            ExecuteA64ExecuteT32(0xD5130501<31:0>); // A64 "MSR DBGDTRTX_EL0,X1"
            else(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
            if !UsingAArch32() then
                X[1] = bits(64) UNKNOWN;
            else(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
                R[1] = bits(32) UNKNOWN;
            // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
            if EDSCR.ERR == '1' then
                EDSCR.RXfull = bit UNKNOWN;
                DBGDTRRX_EL0 = bits(64) UNKNOWN;
            else
                // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
                assert EDSCR.RXfull == '0';

    EDSCR.ITE = '1'; // See comments in EDITR[] (external write)

```

```

return;

return value;
// DBGDTRRX_EL0[] (external read)
// =====

// DBGDTRTX_EL0[] (external write)
// =====bits(32)

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;DBGDTRRX_EL0[boolean memory_mapped]
return DTRRX;

```



```

// DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)
// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

DBGDTR_EL0[] = bits(N) value_in
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>; // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N)bits(32) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0 N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;DBGDTRTX_EL0[boolean_memory_mapped]

    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value; // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1'; // Underrun condition: block side-effects
        return value; // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0'; // See comments in EDITR[] (external write)

    if !UsingAArch32() then
        ExecuteA64(0xB8404401<31:0>); // A64 "LDR W1,[X0],#4"
    else
        ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.TXfull = bit UNKNOWN;
        DBGDTRTX_EL0 = bits(64) UNKNOWN;
    else
        if !UsingAArch32() then

```

```

        ExecuteA64(0xD5130501<31:0>); // A64 "MSR_DBGDTRTX_EL0,X1"
    else
        ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR_DBGDTRTXint,R1"
    // "MSR_DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
    assert EDSCR.TXfull == '1';
    if !UsingAArch32() then
        X[1] = bits(64) UNKNOWN;
    else
        R[1] = bits(32) UNKNOWN;
    EDSCR.ITE = '1'; // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// =====

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;

```

### Library pseudocode for shared/debug/dccanditr/**DTRDBGDTR\_EL0**

```

bits(32) DTRRX;
bits(32) DTRTX; // DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value_in
    bits(N) value = value_in;
    // For MSR_DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR_DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>; // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N)DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0 N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;

```

## Library pseudocode for shared/debug/dccanditr/**EDITRDTR**

```
// EDITR[] (external write)
// =====
// Called on writes to debug register 0x084.bits(32) DTRRX;
bits(32) DTRTX;

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
    return;

    if EDSCR.ERR == '1' then return; // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

    if !Halted() then return; // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.IT0 = '1'; EDSCR.ERR = '1'; // Overrun condition: block write
    return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0> /*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```



```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state// EDITR[] (external write)
// =====
// Called on writes to debug register 0x084.

DCPSInstruction(bits(2) target_el)EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return; // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

    if !

        SynchronizeContextHalted();
    () then return; // Non-debug state: ignore write

    bits(2) handle_el;
    case target_el of
        when if EDSCR.ITE == '0' || EDSCR.MA == '1' then
            EDSCR.ITO = '1'; EDSCR.ERR = '1'; // Overrun condition: block write
            return;

        // ITE indicates whether the processor is ready to accept another instruction; the processor
        // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
        // is no indication that the pipeline is empty (all instructions have completed). In this
        // pseudocode, the assumption is that only one instruction can be executed at a time,
        // meaning ITE acts like "InstrCompl".
        EDSCR.ITE = '0';

    if ! EL1
        if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
            handle_el = PSTATE.EL;
        elsif() then EL2EnabledExecuteA64() && HCR_EL2.TGE == '1' then
            UNDEFINED;
        else
            handle_el =(value);
    else EL1ExecuteT32;
        when EL2
            if !HaveEL(EL2) then
                UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then
                handle_el = EL3;
            elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
                UNDEFINED;
            else
                handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then
                UNDEFINED;
            else
                handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = CurrentSecurityState() == SS_Secure;
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32(); // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTL.R.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2 AArch32.WriteMode(M32_Hyp);
            when EL3AArch32.WriteMode(M32_Monitor);
                if HavePANExt() then

```

```

        if !from_secure then
            PSTATE.PAN = '0';
        elseif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
        if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
        else
            LR = bits(32) UNKNOWN;
            SPSR[] = bits(32) UNKNOWN;
            PSTATE.E = SCTL.R.EE;
            DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

    else // Targeting AArch64
        if UsingAArch32() then
            AArch64.MaybeZeroRegisterUppers();
            PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
            if HavePANExt() && ((handle_el == EL1 && SCTL.R_EL1.SPAN == '0') ||
                (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                HCR_EL2.TGE == '1' && SCTL.R_EL2.SPAN == '0')) then
                PSTATE.PAN = '1';
            ELR[] = bits(64) UNKNOWN; SPSR[] = bits(64) UNKNOWN; ESR[] = bits(64) UNKNOWN;
            DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
            if HaveUA0Ext() then PSTATE.UA0 = '0';
            if HaveMTEExt() then PSTATE.TCO = '1';

            UpdateEDSCRFields(); // Update EDSCR PE state flags
            sync_errors = HaveIESB() && SCTL.R.IESB == '1';
            if HaveDoubleFaultExt() && !UsingAArch32() then
                sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
            // SCTL.R.IESB might be ignored in Debug state.
            if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
                sync_errors = FALSE;
            if sync_errors then
                SynchronizeErrors();
            (value<15:0> /*hw1*/, value<31:16> /*hw2*/);

            EDSCR.ITE = '1';

        return;

```



```

// DRPSInstruction()
// DCPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state // Operation of the DCPS instruction

DRPSInstruction() DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    sync_errors = bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
            else handle_el = EL2;

        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
            handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = IsSecure();
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32(); // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTL.R.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2 AArch32.WriteMode(M32_Hyp);
            when EL3AArch32.WriteMode(M32_Monitor);
                if HavePANExt() then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTL.R.SPAN == '0' then
                        PSTATE.PAN = '1';
            if handle_el == EL2 then
                ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
            else
                LR = bits(32) UNKNOWN;
                SPSR[] = bits(32) UNKNOWN;
                PSTATE.E = SCTL.R[].EE;
                DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

        else // Targeting AArch64
            if UsingAArch32() then
                AArch64.MaybeZeroRegisterUppers();
                PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
                if HavePANExt() && ((handle_el == EL1 && SCTL.R_EL1.SPAN == '0') ||
                    (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                    HCR_EL2.TGE == '1' && SCTL.R_EL2.SPAN == '0')) then
                    PSTATE.PAN = '1';
                ELR[] = bits(64) UNKNOWN; SPSR[] = bits(64) UNKNOWN; ESR[] = bits(64) UNKNOWN;
                DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
                if HaveUA0Ext() then PSTATE.UA0 = '0';
                if HaveMTEExt() then PSTATE.TCO = '1';

    UpdateEDSCRFields(); // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTL.R[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTL.R[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBInDebug) then
        sync_errors = FALSE;

```

```

if sync_errors then
    SynchronizeErrors();

DebugRestorePSR();

return;

```

### Library pseudocode for shared/debug/halting/DebugHaltDRPSInstruction

```

constant bits(6) // DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state DebugHalt_Breakpoint = '000111';
constant bits(6) DRPSInstruction() DebugHalt_EDBGRQ = '010011';
constant bits(6) ();

sync_errors = DebugHalt_Step_Normal = '011011';
constant bits(6) () && DebugHalt_Step_Exclusive = '011111';
constant bits(6) [].IESB == '1';
if DebugHalt_OSUnlockCatch = '100011';
constant bits(6) () && ! DebugHalt_ResetCatch = '100111';
constant bits(6) () then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == DebugHalt_Step_Exclusive);
constant bits(6) ();
// SCTLR[].IESB might be ignored in Debug state.
if ! DebugHalt_HaltInstruction = '101111';
constant bits(6) { DebugHalt_SoftwareAccess = '110011';
constant bits(6) } then
    sync_errors = FALSE;
    if sync_errors then DebugHalt_ExceptionCatch = '110111';
constant bits(6) (); DebugHalt_Step_NoSyndrome = '111011'; ();

return;

```

### Library pseudocode for shared/debug/halting/DebugRestorePSRDebugHalt

```

// DebugRestorePSR()
// =====constant bits(6)

DebugRestorePSR()
// PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
// behave as if UNKNOWN.
if DebugHalt_Breakpoint = '000111';
constant bits(6) UsingAArch32() then
    bits(32) spsr = DebugHalt_EDBGRQ = '010011';
constant bits(6) SPSR[]; DebugHalt_Step_Normal = '011011';
constant bits(6)
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,Q,GE,SS,D,A,I,F> = bits(13) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
else
    bits(64) spsr = DebugHalt_Step_Exclusive = '011111';
constant bits(6) SPSR[]; DebugHalt_OSUnlockCatch = '100011';
constant bits(6)
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN; DebugHalt_ResetCatch = '100111';
constant bits(6)
    DebugHalt_Watchpoint = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess = '110011';
constant bits(6) DebugHalt_ExceptionCatch = '110111';
constant bits(6) UpdateEDSCRFields(); // Update EDSCR PE state flagsDebugHalt_Step_NoSyndrome

```

## Library pseudocode for shared/debug/ halting/DisableITRAndResumeInstructionPrefetchDebugRestorePSR

```
// DebugRestorePSR()
// =====

DebugRestorePSR()
// PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
// behave as if UNKNOWN.
if UsingAArch32() then
    bits(32) spsr = SPSR[];
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
else
    bits(64) spsr = SPSR[];
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
    UpdateEDSCRFieldsDisableITRAndResumeInstructionPrefetch(); // Update
```

## Library pseudocode for shared/debug/ halting/ExecuteA64DisableITRAndResumeInstructionPrefetch

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);DisableITRAndResumeInstructionPrefetch();
```

## Library pseudocode for shared/debug/halting/ExecuteT32ExecuteA64

```
// Execute a T32 instruction in Debug state. // Execute an A64 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);ExecuteA64(bits(32) instr);
```

## Library pseudocode for shared/debug/halting/ExitDebugStateExecuteT32

```
// ExitDebugState()
// =====// Execute a T32 instruction in Debug state.

ExitDebugState()
    assertExecuteT32(bits(16) hw1, bits(16) hw2); Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001'; // Signal restarting
    // Clear any pending Halting debug events
    if Havev8p8Debug() then
        EDESR<3:0> = '0000';
    else
        EDESR<2:0> = '000';

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR);
        spsr = ZeroExtend(DSPSR);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    if UsingAArch32() then
        SetPSTATEFromPSR(spsr<31:0>); // Can update privileged bits, even at EL0
    else
        SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

    boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
            new_pc<63:32> = Zeros();
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

    (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```



```

// Halt()
// =====// ExitDebugState()
// =====

Halt(bits(6) reason)
    boolean is_async = FALSE;ExitDebugState()
    assert
        HaltHalted(reason, is_async);

// Halt()
// =====();

Halt(bits(6) reason, boolean is_async)();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001'; // Signal restarting
    EDESR<2:0> = '000'; // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(64) spsr;

    if

        CTI_SignalEventUsingAArch32(()) then
            new_pc = CrossTriggerIn_CrossHaltZeroExtend(); // Trigger other cores to halt

        bits(64) preferred_restart_address = (DLR);
        spsr = ThisInstrAddrZeroExtend();
        bits(32) spsr_32;
        bits(64) spsr_64;
    (DSPSR);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
        // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
        if UsingAArch32() then
            spsr_32 = () then GetPSRFromPSTATESetPSTATEFromPSR({spsr<31:0>}); // Can update pr
        else DebugStateSetPSTATEFromPSR();
        else
            spsr_64 = (spsr); // Can update privileged bits, even at EL0

        boolean branch_conditional = FALSE;
        if GetPSRFromPSTATEUsingAArch32(()) then
            if DebugStateConstrainUnpredictableBool();

        if ((HaveBTIExtUnpredictable_RESTARTALIGNPC() && !is_async && !(reason IN {}) then new_pc<0> = '0';
        // AArch32 branch DebugHalt_Step_NormalBranchTo, (new_pc<31:0>, DebugHalt_Step_ExclusiveBranchType
        else
            // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
            if spsr<4> == '1' &&
                DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
                ConstrainUnpredictableBool(Unpredictable_ZEROBTTYPEUnpredictable_RESTARTZEROUPPERPC)) then
            if) then
                new_pc<63:32> = UsingAArch32Zeros() then
                spsr_32<11:10> = '00';
            else
                spsr_64<11:10> = '00';

        if();
        // A type of branch that is never predicted UsingAArch32BranchTo() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr_32;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr_64;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if(new_pc, CurrentSecurityStateBranchType_DBGEXIT() ==, branch_conditional);

```

```

(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1'); // Atomically signal restarted SS_SecureUpdateEDS
    EDSCR.SDD = '0'; // If entered in Secure state, allow debug
elseif(); // Stop signalling PE state HaveELDisableITRAndResumeInstruct
    EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
else
    EDSCR.SDD = '1'; // Otherwise EDSCR.SDD is RES1
EDSCR.MA = '0';

// In Debug state:
// * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
// * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
// are not changed on exception entry, this function also leaves them unchanged.
// * PSTATE.{IT,T} are ignored.
// * PSTATE.IL is ignored and behave-as-if 0.
// * PSTATE.BTYPE is ignored and behave-as-if 0.
// * PSTATE.TCO is set 1.
// * PSTATE.{UA0,PAN} are observable and not changed on entry into Debug state.
if UsingAArch32() then
    PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
else
    PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;
    PSTATE.TCO = '1';
    PSTATE.BTYPE = '00';
    PSTATE.IL = '0';

StopInstructionPrefetchAndEnableITR();
EDSCR.STATUS = reason; // Signal entered Debug state
UpdateEDSCRFields(); // Update EDSCR PE state flags.
());

return;

```

## Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpointHalt

```
// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean// Halt()
// ===== HaltOnBreakpointOrWatchpoint()
returnHalt(bits(6) reason) (CrossTriggerIn_CrossHalt); // Trigger other cores to halt

bits(64) preferred_restart_address = ThisInstrAddr();
bits(32) spsr_32;
bits(64) spsr_64;
if UsingAArch32() then
    spsr_32 = GetPSRFromPSTATE(DebugState);
else
    spsr_64 = GetPSRFromPSTATE(DebugState);

if (HaveBTIExt() &&
    !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive, DebugHalt_Step_NoSyndrome,
    DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
    ConstrainUnpredictableBool(Unpredictable_ZEROBTYP)) then
    if UsingAArch32() then
        spsr_32<11:10> = '00';
    else
        spsr_64<11:10> = '00';

if UsingAArch32() then
    DLR = preferred_restart_address<31:0>;
    DSPSR = spsr_32;
else
    DLR_EL0 = preferred_restart_address;
    DSPSR_EL0 = spsr_64;

EDSCR.ITE = '1';
EDSCR.IT0 = '0';
if IsSecure() then
    EDSCR.SDD = '0'; // If entered in Secure state, allow debug
elseif HaveEL(EL3) then
    EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
else
    assert EDSCR.SDD == '1'; // Otherwise EDSCR.SDD is RES1
EDSCR.MA = '0';

// In Debug state:
// * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
// * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
//   are not changed on exception entry, this function also leaves them unchanged.
// * PSTATE.{IT,T} are ignored.
// * PSTATE.IL is ignored and behave-as-if 0.
// * PSTATE.BTYPE is ignored and behave-as-if 0.
// * PSTATE.TCO is set 1.
// * PSTATE.{UA0,PAN} are observable and not changed on entry into Debug state.
if UsingAArch32() then
    PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
else
    PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;
    PSTATE.TCO = '1';
    PSTATE.BTYPE = '00';
PSTATE.IL = '0';

StopInstructionPrefetchAndEnableITR();
EDSCR.STATUS = reason; // Signal entered Debug state
UpdateEDSCRFieldsHaltingAllowedCTI_SignalEvent() && EDSCR.HDE == '1' && OSLR_EL1.OSLK == '0';();

return;
```

### Library pseudocode for shared/debug/halting/**HaltedHaltOnBreakpointOrWatchpoint**

```
// Halted()
// =====
// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'}); // HaltedHaltOnBreakpointOrWatchpoint
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

### Library pseudocode for shared/debug/halting/**HaltingAllowedHalted**

```
// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.
// Halted()
// =====

boolean HaltingAllowed()
    if Halted()
        return !(EDSCR.STATUS IN {'000001', '000010'}); // Halted Halted() || DoubleLockStatus()
    return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure return ExternalSecureInvasiveDebugEnabled();
```

### Library pseudocode for shared/debug/halting/**RestartingHaltingAllowed**

```
// Restarting()
// =====
// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean Restarting()
    return EDSCR.STATUS == '000001'; // RestartingHaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = SecurityStateAtEL(PSTATE.EL);
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure return ExternalSecureInvasiveDebugEnabled();
```

### Library pseudocode for shared/debug/halting/**StopInstructionPrefetchAndEnableITRRestarting**

```
// Restarting()
// =====

boolean StopInstructionPrefetchAndEnableITR(); Restarting()
    return EDSCR.STATUS == '000001'; // Restarting
```

**Library pseudocode for shared/debug/  
halting/UpdateEDSCRFieldsStopInstructionPrefetchAndEnableITR**

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        ss = CurrentSecurityState();
        EDSCR.NS = if ss == SS_Secure then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;

    return;StopInstructionPrefetchAndEnableITR();
```

## Library pseudocode for shared/

### debug/haltingevents/halting/CheckExceptionCatchUpdateEDSCRFields

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level// UpdateEDSCRFields
// =====
// Update EDSCR PE state fields

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    ss = UpdateEDSCRFields()

    if ! Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        ss = SecurityStateAtEL(PSTATE.EL);
        base = if ss == EDSCR.NS = if ss == SS_Secure then 0 else 4;
        if then '0' else '1';

        bits(4) RW;
        RW<1> = if HaltingAllowedELUsingAArch32() then
            boolean halt;
            if( HaveExtendedECDebugEventsEL1() then
                exception_exit = !exception_entry;
                increment = 8;
                ctrl = EDECCR< then '0' else '1';
            if PSTATE.EL != UIntEL0(PSTATE.EL) + base + increment>:EDECCR< then
                RW<0> = RW<1>;
            else
                RW<0> = if UIntUsingAArch32(PSTATE.EL) + base>;
                case ctrl of
                    when '00' halt = FALSE;
                    when '01' halt = TRUE;
                    when '10' halt = (exception_exit == TRUE);
                    when '11' halt = (exception_entry == TRUE);
                else
                    halt = (EDECCR<() then '0' else '1';
                if ! UIntHaveEL(PSTATE.EL) + base> == '1');

            if halt then
                if( Havev8p8DebugEL2() && exception_entry then
                    EDESR.EC = '1';
                else) || (
                    HaltHaveEL() && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
                    RW<2> = RW<1>;
                else
                    RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
                if !HaveEL(EL3) then
                    RW<3> = RW<2>;
                else
                    RW<3> = if ELUsingAArch32(EL3DebugHalt_ExceptionCatchEL3);) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

## Library pseudocode for shared/debug/haltingevents/CheckHaltingStepCheckExceptionCatch

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

CheckHaltingStep(boolean is_async)
    ifCheckExceptionCatch(boolean exception_entry)
        // Called after an exception entry or exit, that is, such that the Security state
        // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
        // is not implemented, this function might also be called at any time.
        ss = HaltingAllowedSecurityStateAtEL() && EDESR.SS == '1' then
            // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
            if(PSTATE.EL);
            base = if ss == HaltingStep_DidNotStepSS_Secure() then 0 else 4;
            if
                HaltHaltingAllowed(()) then
                    boolean halt;
                    ifDebugHalt_Step_NoSyndromeHaveExtendedECDebugEvents, is_async);
                    elseif() then
                        exception_exit = !exception_entry;
                        increment = 8;
                        ctrl = EDECCR< HaltingStep_SteppedEXUInt() then (PSTATE.EL) + base + increment>;EDECCR<
                        HaltUInt((PSTATE.EL) + base>;
                        case ctrl of
                            when '00' halt = FALSE;
                            when '01' halt = TRUE;
                            when '10' halt = (exception_exit == TRUE);
                            when '11' halt = (exception_entry == TRUE);
                        else
                            halt = (EDECCR<DebugHalt_Step_ExclusiveUInt, is_async);
                        else(PSTATE.EL) + base> == '1');
                        if halt then
                            Halt(DebugHalt_Step_NormalDebugHalt_ExceptionCatch, is_async);};
```

## Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatchCheckHaltingStep

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckOSUnlockCatch()
    if ((CheckHaltingStep()
    ifHaveDoPDHaltingAllowed() && CTIDEVCTL.OSUCE == '1') ||
        (!() && EDESR.SS == '1' then
            // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
            ifHaveDoPDHaltingStep_DidNotStep() && EDECR.OSUCE == '1')) then
                if !() then(DebugHalt_Step_NoSyndrome);
                elseif HaltingStep_SteppedEX() then
                    Halt(DebugHalt_Step_Exclusive);
                else
                    Halt(DebugHalt_Step_NormalHaltedHalt() then EDESR.OSUC = '1');};
```

## Library pseudocode for shared/debug/haltingevents/CheckPendingExceptionCatchCheckOSUnlockCatch

```
// CheckPendingExceptionCatch()
// =====
// Check whether EDESR.EC has been set by an Exception Catch debug event.// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckPendingExceptionCatch(boolean is_async)
    if CheckOSUnlockCatch()
        if (( Havev8p8DebugHaveDoPD() &&() && CTIDEVCTL.OSUCE == '1') ||
            (! HaltingAllowedHaveDoPD() && EDESR.EC == '1' then() && EDECR.OSUCE == '1')) then
            if !
                HaltHalted(DebugHalt_ExceptionCatch, is_async);() then EDESR.OSUC = '1';
```

## Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        boolean is_async = TRUE;() && EDESR.OSUC == '1' then
            Halt(DebugHalt_OSUnlockCatch, is_async);;
```

## Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        boolean is_async = TRUE;() && EDESR.RC == '1' then
            Halt(DebugHalt_ResetCatch, is_async);;
```

## Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDECR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

## Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed() then
        boolean is_async = TRUE;() then
            Halt(DebugHalt_EDBGRO, is_async);
);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

## Library pseudocode for shared/debug/haltingevents/HaltingStep\_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

## Library pseudocode for shared/debug/haltingevents/HaltingStep\_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

## Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// =====

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted(); // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elseif active && HaltingAllowed() then
        boolean advance;
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

## Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    SecurityState ss = SecurityStateAtEL(target);
    if Havev8p4Debug() then
        if EDSCR.INTdis[0] == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure    int_dis = ExternalSecureInvasiveDebugEnabled();
            else
                int_dis = FALSE;
        else
            case target of
                when EL3
                    int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
                when EL2
                    int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled());
                when EL1
                    if ss == SS_Secure then
                        int_dis = (EDSCR.INTdis IN {'1x'} && ExternalSecureInvasiveDebugEnabled());
                    else
                        int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
            return int_dis;
```

## Library pseudocode for shared/debug/pmu/GetNumEventCounters

```
// GetNumEventCounters()
// =====
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";
```

## Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.
boolean HasElapsed64Cycles();
```

## Library pseudocode for shared/debug/pmu/PMUCountValue

```
// PMUCountValue()
// =====
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by, if the event it is
// configured to count yields the value 'V' on this cycle.

integer PMUCountValue(integer n, integer V)
    if !HavePMUv3TH() then
        return V;

    integer T = UInt(PMEVTYPER_EL0[n].TH);

    case PMEVTYPER_EL0[n].TC of
        when '000' return (if V != T then V else 0); // Disabled or not-equal
        when '001' return (if V != T then 1 else 0); // Not-equal, count
        when '010' return (if V == T then V else 0); // Equals
        when '011' return (if V == T then 1 else 0); // Equals, count
        when '100' return (if V >= T then V else 0); // Greater-than-or-equal
        when '101' return (if V >= T then 1 else 0); // Greater-than-or-equal, count
        when '110' return (if V < T then V else 0); // Less-than
        when '111' return (if V < T then 1 else 0); // Less-than, count
```

## Library pseudocode for shared/debug/pmu/PMUCounterMask

```
constant integer CYCLE_COUNTER_ID = 31;

// PMUCounterMask()
// =====
// Return bitmask of accessible PMU counters.

bits(32) PMUCounterMask()
    integer n;
    if UsingAArch32() then
        n = AArch32.GetNumEventCountersAccessible();
    else
        n = AArch64.GetNumEventCountersAccessible();
    return '1' : ZeroExtend(Ones(n), 31);
```

## Library pseudocode for shared/debug/pmu/PMUEvent

```
// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1. constant bits(16)

PMUEvent(bits(16) event) PMU_EVENT_SW_INCR = 0x0000<15:0>;
constant bits(16)
    PMU_EVENT_INST_RETIRED = 0x0008<15:0>;
constant bits(16) PMU_EVENT_EXC_TAKEN = 0x0009<15:0>;
constant bits(16) PMU_EVENT_CPU_CYCLES = 0x0011<15:0>;
constant bits(16) PMU_EVENT_INST_SPEC = 0x001B<15:0>;
constant bits(16) PMU_EVENT_CHAIN = 0x001E<15:0>;

// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) event)
    PMUEvent(event, 1);

// PMUEvent()
// =====
// Accumulate a PMU Event.

PMUEvent(bits(16) event, integer increment)
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(event, increment, idx);

// PMUEvent()
// =====
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) event, integer increment, integer idx)
    if !HavePMUv3() then
        return;

    if UsingAArch32() then
        if PMEVTYPER[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPER_EL0[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```

## Library pseudocode for shared/debug/pmu/integer

```
array integer PMUEventAccumulator[0..30]; // Accumulates PMU events for a cycle
```

## Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// =====

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = pc_sample.ns = if CurrentSecurityStateIsSecure();
    () then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if (HaveVirtHostExt() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
return;
```

## Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =====

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

bits(32) sample;
if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        if HaveVirtHostExt() && EDSCR.SC2 == '1' then
            EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            EDPCSRhi.EL = pc_sample.el;
            EDPCSRhi.NS = (if pc_sample.ss == pc_sample.ns then pc_sample.ns else pc_sample.ns);
        else
            EDPCSRhi = (if pc_sample.rw == '0' then SS_Secure then '0' else '1');
        else
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
                EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                    else bits(32) UNKNOWN);
            else
                EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                    then pc_sample.vmid else Zeros());
                EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
            );
            EDVIDSR.NS = pc_sample.ns;
            EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
            EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
            // The conditions for setting HV are not specified if PCSRhi is zero.
            // An example implementation may be "pc_sample.rw".
            EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
        else
            sample = Ones(32);
            if update then
                EDPCSRhi = bits(32) UNKNOWN;
                EDCIDSR = bits(32) UNKNOWN;
                EDVIDSR = bits(32) UNKNOWN;

return sample;
```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (  
    boolean valid,  
    bits(64) pc,  
    bits(2) el,  
    bit rw, bit rw,  
    bit ns,  
    boolean has_el2,  
    bits(32) contextidr,  
    bits(32) contextidr_el2,  
    boolean el0h,  
    bits(16) vmid  
)  
  
    SecurityState ss,  
    boolean has_el2,  
    bits(32) contextidr,  
    bits(32) contextidr_el2,  
    boolean el0h,  
    bits(16) vmid  
)  
  
PCSample pc_sample;
```

## Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)  
// =====  
  
bits(32) PMPCSR[boolean memory_mapped]  
    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits  
        IMPLEMENTATION_DEFINED "generate error response";  
        return bits(32) UNKNOWN;  
  
    // The Software lock is OPTIONAL.  
    update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects  
  
    bits(32) sample;  
    if pc_sample.valid then  
        sample = pc_sample.pc<31:0>;  
        if update then  
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);  
            PMPCSR.EL = pc_sample.el;  
            PMPCSR.NS = (if pc_sample.ss == PMPCSR.NS = pc_sample.ns;  
  
            PMCID1SR = pc_sample.contextidr;  
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;  
  
            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN { SS_Secure then '0' else '1'};  
  
            PMCID1SR = pc_sample.contextidr;  
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;  
  
            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h  
                            then pc_sample.vmid else bits(16) UNKNOWN);  
        else  
            sample = Ones(32);  
            if update then  
                PMPCSR<55:32> = bits(24) UNKNOWN;  
                PMPCSR.EL = bits(2) UNKNOWN;  
                PMPCSR.NS = bit UNKNOWN;  
  
                PMCID1SR = bits(32) UNKNOWN;  
                PMCID2SR = bits(32) UNKNOWN;  
  
                PMVIDSR.VMID = bits(16) UNKNOWN;  
  
    return sample;
```

## Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS == 1
    if step_enabled && PSTATE.SS == '0' then
        AArch64.SoftwareStepException();
```

## Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    if UsingAArch32() then
        assert N == 32;
    else
        assert N == 64;

    assert Halted() || Restarting() || PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    boolean valid;
    bits(2) dest_el;
    bits(2) dest;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
        dest = PSTATE.EL;
    else
        (valid, dest_el) = (valid, dest) = ELFromSPSR(spsr); assert valid;

    dest_ss = dest_is_secure = () || dest == EL3SecurityStateAtELIsSecureBelowEL3(dest_el);
    ;

    bit mask;
    boolean enabled_at_dest;
    dest_using_32 = (if dest_el == EL3 then dest_using_32 = (if dest == EL0 then spsr<4> == '1' else ELUsingAArch32(dest));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el, dest_ss);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;
```

## Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// =====
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception. Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.
boolean SoftwareStep_DidNotStep();
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_SteppedEX

```
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test. Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();
```

## Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            // * CV set to 0 and COND is set to an UNKNOWN value
            // * CV set to 1 and COND is set to the condition code for the condition that
            //   applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

## Library pseudocode for shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,      // Uncategorized or unknown reason
                        Exception_WFxTrap,           // Trapped WFI or WFE instruction
                        Exception_CP15RTTTrap,        // Trapped AArch32 MCR or MRC access, coproc=0b1110
                        Exception_CP15RRTTrap,        // Trapped AArch32 MCRR or MRRC access, coproc=0b1110
                        Exception_CP14RTTTrap,        // Trapped AArch32 MCR or MRC access, coproc=0b1110
                        Exception_CP14DTTTrap,        // Trapped AArch32 LDC or STC access, coproc=0b1110
                        Exception_CP14RRTTrap,        // Trapped AArch32 MRRC access, coproc=0b1110
                        Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                        Exception_FPIDTrap,           // Trapped access to SIMD or FP ID register
                        Exception_LDST64BTrap,        // Trapped access to ST64BV, ST64BV0, ST64B and LD
                        // Trapped BXJ instruction not supported in Armv8
                        Exception_PACTrap,            // Trapped invalid PAC use
                        Exception_IllegalState,       // Illegal Execution state
                        Exception_SupervisorCall,     // Supervisor Call
                        Exception_HypervisorCall,     // Hypervisor Call
                        Exception_MonitorCall,        // Monitor Call or Trapped SMC instruction
                        Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
                        Exception_ERetTrap,           // Trapped invalid ERET use
                        Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                        Exception_PCAAlignment,        // PC alignment fault
                        Exception_DataAbort,          // Data Abort
                        Exception_NV2DataAbort,       // Data abort at EL1 reported as being from EL2
                        Exception_PACFail,            // PAC Authentication failure
                        Exception_SPAlignment,        // SP alignment fault
                        Exception_FPTrappedException, // IEEE trapped FP exception
                        Exception_SError,             // SError interrupt
                        Exception_Breakpoint,         // (Hardware) Breakpoint
                        Exception_SoftwareStep,       // Software Step
                        Exception_Watchpoint,         // Watchpoint
                        Exception_NV2Watchpoint,      // Watchpoint at EL1 reported as being from EL2
                        Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
                        Exception_VectorCatch,        // AArch32 Vector Catch
                        Exception_IRQ,                // IRQ interrupt
                        Exception_BranchTarget,       // Branch Target Identification
                        Exception_MemCpyMemSet,       // Exception from a CPY* or SET* instruction
                        Exception_FIQ};               // FIQ interrupt
```

## Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (
    Exception exceptype,      // Exception class
    bits(25) syndrome,       // Syndrome record
    bits(5) syndrome2,       // ST64BV(0) return value register specifier
    bits(64) vaddress,       // Virtual fault address
    boolean ipavalid,        // Validity of Intermediate Physical fault address
    bit NS,                  // Intermediate Physical fault address space
    bits(52) ipaddress,      // Intermediate Physical fault address
    boolean trappedsyscallinst) // Trapped SVC or SMC instruction
```

## Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.syndrome2 = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();
    r.trappedsyscallinst = FALSE;
    return r;
```

## Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    if level == -1 then
        assert Have52BitIPAAAndPASpaceExt();
        case statuscode of
            when Fault_AddressSize          result = '101001';
            when Fault_Translation          result = '101011';
            when Fault_SyncExternalOnWalk    result = '010011';
            when Fault_SyncParityOnWalk      result = '011011'; assert !HaveRASExt();
            otherwise                        Unreachable();

        return result;
    case statuscode of
        when Fault_AddressSize          result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag           result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Permission            result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation           result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal          result = '010000';
        when Fault_SyncExternalOnWalk    result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity            result = '011000';
        when Fault_SyncParityOnWalk      result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity           result = '011001';
        when Fault_AsyncExternal         result = '010001';
        when Fault_Alignment             result = '100001';
        when Fault_Debug                 result = '100010';
        when Fault_TLBConflict           result = '110000';
        when Fault_HWUpdateAccessFlag    result = '110001';
        when Fault_Lockdown              result = '110100'; // IMPLEMENTATION DEFINED
        when Fault_Exclusive             result = '110101'; // IMPLEMENTATION DEFINED
        otherwise                        Unreachable();

    return result;
```

## Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.s2fslwalk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
```

## Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

## Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =====
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =====
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```

## Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// =====

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

## Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
// <10> - Syndrome valid bit. The syndrome is only valid for certain types of access instruction.
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.
bits(11) LSInstructionSyndrome();
```

### Library pseudocode for shared/functions/cache/CACHE\_OP

```
// CACHE_OP()
// =====
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

### Library pseudocode for shared/functions/cache/CPASAtPAS

```
// CPASAtPAS()
// =====
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS\_NonSecure
            return CPAS\_NonSecure;
        when PAS\_Secure
            return CPAS\_Secure;
```

### Library pseudocode for shared/functions/cache/CPASAtSecurityState

```
// CPASAtSecurityState()
// =====
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS\_NonSecure
            return CPAS\_NonSecure;
        when SS\_Secure
            return CPAS\_SecureNonSecure;
```

### Library pseudocode for shared/functions/cache/CacheOp

```
enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

### Library pseudocode for shared/functions/cache/CacheOpScope

```
enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

### Library pseudocode for shared/functions/cache/CachePASpace

```
enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_SecureNonSecure,    // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

## Library pseudocode for shared/functions/cache/CacheRecord

```
type CacheRecord is (
  AccType      acctype,          // Access type
  CacheOp      cacheop,         // Cache operation
  CacheOpScope opscope,         // Cache operation type
  CacheType    cachetype,       // Cache type
  bits(64)     regval,
  FullAddress  paddress,
  bits(64)     vaddress,        // For VA operations
  integer      set,             // For SW operations
  integer      way,             // For SW operations
  integer      level,           // For SW operations
  Shareability shareability,
  boolean      translated,
  boolean      is_vmid_valid,    // is vmid valid for current context
  bits(16)     vmid,
  boolean      is_asid_valid,    // is asid valid for current context
  bits(16)     asid,
  SecurityState security,
  // For cache operations to full cache or by set/way
  // For operations by address, PA space in paddress
  CachePASpace cpas
)
```

## Library pseudocode for shared/functions/cache/CacheType

```
enumeration CacheType {
  CacheType_Data,
  CacheType_Tag,
  CacheType_Data_Tag,
  CacheType_Instruction
};
```

## Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// =====
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
  if CLIDR_EL1.LoC == '000' then
    return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoC is before any"

  if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
    return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoU is before any"

  return TRUE;
```

## Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// =====
// Decode input value into set, way and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
  level = UInt(regval[3:1]);
  (set, way, linesize) = GetCacheInfo(level, cachetype);
  return (set, way, level);
```

## Library pseudocode for shared/functions/cache/GetCacheInfo

```
// Returns numsets, associativity & linesize.
(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

### Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

```
// ICInstNeedsTranslation()
// =====
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";
```

### Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/ASR\_C

```
// ASR_C()
// =====

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

### Library pseudocode for shared/functions/common/Align

```
// Align()
// =====

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =====

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

## Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// =====

integer BitCount(bits(N) x)
  integer result = 0;
  for i = 0 to N-1
    if x<i> == '1' then
      result = result + 1;
  return result;
```

## Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
  return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

## Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// =====

integer CountLeadingZeroBits(bits(N) x)
  return N - (HighestSetBit(x) + 1);
```

## Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e, integer size]
  assert e >= 0 && (e+1)*size <= N;
  return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e]
  return Elem[vector, e, size];

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e, integer size] = bits(size) value
  assert e >= 0 && (e+1)*size <= N;
  vector<(e+1)*size-1:e*size> = value;
  return;

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e] = bits(size) value
  Elem[vector, e, size] = value;
  return;
```

### Library pseudocode for shared/functions/common/Extend

```
// Extend()
// =====

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// =====

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

### Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// =====

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

### Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

### Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

### Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

### Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

### Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/LSL\_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/LSR\_C

```
// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// =====

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

### Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

### Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

### Library pseudocode for shared/functions/common/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// =====

bits(N) Ones()
    return Ones(N);
```

### Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/ROR\_C

```
// ROR_C()
// =====

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// =====

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

### Library pseudocode for shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

### Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// =====

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

### Library pseudocode for shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

### Library pseudocode for shared/functions/common/SInt

```
// SInt()
// =====

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

### Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// =====

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// =====

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

### Library pseudocode for shared/functions/common/Split64to32

```
// Split64to32()
// =====

(bits(32), bits(32)) Split64to32(bits(64) value)
    return (value<63:32>, value<31:0>);
```

### Library pseudocode for shared/functions/common/UInt

```
// UInt()
// =====

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

### Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

### Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0', N);

// Zeros()
// =====

bits(N) Zeros()
    return Zeros(N);
```

## Library pseudocode for shared/functions/counters/AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// =====
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();

    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S,
                                           CNTP_CTL_S.IMASK, InterruptID_CNTPS);
            CNTP_CTL_S.ISTATUS = if status then '1' else '0';

        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS,
                                           CNTP_CTL_NS.IMASK, InterruptID_CNTP);
            CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL,
                                           CNTP_CTL.IMASK, InterruptID_CNTP);
            CNTP_CTL.ISTATUS = if status then '1' else '0';

    if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL,
                                       CNTHP_CTL.IMASK, InterruptID_CNTHP);
        CNTHP_CTL.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                       CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    return;
```

## Library pseudocode for shared/functions/counters/AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// =====
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    boolean ecv = FALSE;
    if HaveECVExt() then
        ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
    if ecv then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);

    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, CNTP_CTL_EL0.IMASK, InterruptID_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
        CNTHP_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                                     CNTHP_CTL_EL2.IMASK, InterruptID_CNTHP);
        CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && HaveSecureEL2Ext() && CNTHPS_CTL_EL2.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                     CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTPS_CTL_EL1.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1,
                                     CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS);
        CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        imask = CNTV_CTL_EL0.IMASK;
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
        CNTHV_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                                     CNTHV_CTL_EL2.IMASK, InterruptID_CNTHV);
        CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';

    if ((HaveSecureEL2Ext() && HaveVirtHostExt()) &&
        CNTHVS_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                                     CNTHVS_CTL_EL2.IMASK, InterruptID_CNTHVS);
        CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
    return;
```

## Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if HaveCNTSCEExt() && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

## Library pseudocode for shared/functions/counters/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                           bits(1) imask, InterruptID intid)
    boolean conditon_met;
    signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) -
                    UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

## Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

## Library pseudocode for shared/functions/counters/SetEventRegister

```
// SetEventRegister()
// =====
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/counters/TestEventCNTP

```
// TestEventCNTP()
// =====
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
  bits(64) offset;
  bits(1) samplebit, previousbit;
  if CNTHCTL_EL2.EVNTEN == '1' then
    n = UInt(CNTHCTL_EL2.EVNTI);
    if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
      n = n + 8;
    boolean ecv = FALSE;
    if HaveECVExt() then
      ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
        SCR_EL3.ECVEn == '1');
      offset = if ecv then CNTPOFF_EL2 else Zeros(64);
    samplebit = (current_physical_count - offset)<n>;
    previousbit = (prev_physical_count - offset)<n>;
    if CNTHCTL_EL2.EVNTDIR == '0' then
      if previousbit == '0' && samplebit == '1' then SetEventRegister();
    else
      if previousbit == '1' && samplebit == '0' then SetEventRegister();
  return;
```

## Library pseudocode for shared/functions/counters/TestEventCNTV

```
// TestEventCNTV()
// =====
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
  bits(64) offset;
  bits(1) samplebit, previousbit;
  if (!(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
    CNTKCTL_EL1.EVNTEN == '1') then
    n = UInt(CNTKCTL_EL1.EVNTI);
    if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
      n = n + 8;
    if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
      offset = CNTVOFF_EL2;
    else
      offset = Zeros(64);
    samplebit = (current_physical_count - offset)<n>;
    previousbit = (prev_physical_count - offset)<n>;
    if CNTKCTL_EL1.EVNTDIR == '0' then
      if previousbit == '0' && samplebit == '1' then SetEventRegister();
    else
      if previousbit == '1' && samplebit == '0' then SetEventRegister();
  return;
```

## Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// =====

bits(N) BitReverse(bits(N) data)
  bits(N) result;
  for i = 0 to N-1
    result<(N-i)-1> = data<i>;
  return result;
```

## Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()  
// =====  
  
boolean HaveCRCExt()  
    return HasArchVersion\(ARMv8p1\) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

## Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()  
// =====  
  
// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation  
  
bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)  
    assert N > 32;  
    bits(N) data = data_in;  
    for i = N-1 downto 32  
        if data<i> == '1' then  
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));  
    return data<31:0>;
```

## Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()  
// =====  
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.  
  
bits(128) AESInvMixColumns(bits (128) op)  
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;  
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;  
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;  
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;  
  
    bits(4*8) out0;  
    bits(4*8) out1;  
    bits(4*8) out2;  
    bits(4*8) out3;  
  
    for c = 0 to 3  
        out0<c*8+:8> = FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR FFmul09(in3<c*8+:8>);  
        out1<c*8+:8> = FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0D(in3<c*8+:8>);  
        out2<c*8+:8> = FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR FFmul0B(in3<c*8+:8>);  
        out3<c*8+:8> = FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR FFmul0E(in3<c*8+:8>);  
  
    return (  
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :  
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :  
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :  
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>  
    );
```

## Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()  
// =====  
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.  
  
bits(128) AESInvShiftRows(bits(128) op)  
    return (  
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :  
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :  
        op< 95: 88> : op<119:112> : op< 15:  8> : op< 39: 32> :  
        op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0>  
    );
```

## Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
  // Inverse S-box values
  bits(16*16*8) GF2_inv = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
    /*E*/ 0x619953833cbbefbc8b0f52aae4d3be0a0<127:0> :
    /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
    /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
    /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
    /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
    /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
    /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
    /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
    /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
    /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
    /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
    /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
    /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
    /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
    /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
  );
  bits(128) out;
  for i = 0 to 15
    out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
  return out;
```

## Library pseudocode for shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// =====
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
  bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
  bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
  bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
  bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

  bits(4*8) out0;
  bits(4*8) out1;
  bits(4*8) out2;
  bits(4*8) out3;

  for c = 0 to 3
    out0<c*8+:8> = FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR          in2<c*8+:8> EOR
    out1<c*8+:8> =          in0<c*8+:8> EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
    out2<c*8+:8> =          in0<c*8+:8> EOR          in1<c*8+:8> EOR FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>)
    out3<c*8+:8> = FFmul03(in0<c*8+:8>) EOR          in1<c*8+:8> EOR          in2<c*8+:8> EOR FFmul02(in3<c*8+:8>)

  return (
    out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
    out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
    out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
    out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
  );
```

## Library pseudocode for shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// =====
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 95: 88> : op< 55: 48> : op< 15:  8> : op<103: 96> :
        op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
        op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
        op<127:120> : op< 87: 80> : op< 47: 40> : op<  7:  0>
    );
```

## Library pseudocode for shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =====
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

## Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =====

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAF9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBBCBAB8B6B4B2B0AEACAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =====

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =====

bits(8) FFmul09(bits(8) b)
  bits(256*8) FFmul_09 = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
    /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
    /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
    /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
    /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
    /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
    /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
    /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
    /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
    /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
    /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
    /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
    /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
    /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
    /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
    /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
  );
  return FFmul_09<UInt(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =====

bits(8) FFmul0B(bits(8) b)
  bits(256*8) FFmul_0B = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
    /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
    /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
    /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
    /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
    /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
    /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
    /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
    /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
    /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
    /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
    /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
    /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
    /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
    /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
    /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
  );
  return FFmul_0B<UInt(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =====

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBCECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADD00<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =====

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDFAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// =====
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// =====
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =====
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// =====
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =====
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion\(ARMv8p2\) || !(HaveSHA1Ext\(\) && HaveSHA256Ext\(\)) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// =====
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion\(ARMv8p2\) || !(HaveSHA1Ext\(\) && HaveSHA256Ext\(\)) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// =====
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// =====
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

### Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
        maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
        t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
        x<127:96> = t + x<127:96>;
        y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
        <y, x> = ROL(y : x, 32);
    return (if part1 then x else y);
```

### Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// =====

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return ((y EOR z) AND x) EOR z;
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// =====

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// =====

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

### Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =====

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

### Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// =====

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

### Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// =====
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e
    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

### Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

### Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// =====
// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

### Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

### Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at address.
boolean IsExclusiveGlobal(FullAddress address, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at address.
boolean IsExclusiveLocal(FullAddress address, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at address in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress address, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at address in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress address, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

### Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// =====

boolean AArch32.HaveHPDExt()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 hierarchical permission disables");
```

### Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// =====

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

## Library pseudocode for shared/functions/extension/Have16bitVMID

```
// Have16bitVMID()
// =====
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return (HasArchVersion(ARMv8p1) && HaveEL(EL2) &&
        boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID");
```

## Library pseudocode for shared/functions/extension/Have52BitIPAAndPASpaceExt

```
// Have52BitIPAAndPASpaceExt()
// =====
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
    return (HasArchVersion(ARMv8p7) &&
        boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
        Have52BitVAExt() && Have52BitPAExt());
```

## Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// =====
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support");
```

## Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// =====
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support");
```

## Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// =====
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension");
```

## Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// =====
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension");
```

## Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateExtHaveAltFP

```
// HaveAccessFlagUpdateExt()
// =====
// HaveAltFP()
// =====
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAccessFlagUpdateExt()
HaveAltFP()
    return HasArchVersion(ARMv8p1ARMv8p7);
```

## Library pseudocode for shared/functions/extension/HaveAltFPHaveAtomicExt

```
// HaveAltFP()
// =====
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.
// HaveAtomicExt()
// =====

boolean HaveAltFP()
HaveAtomicExt()
    return HasArchVersion(ARMv8p7ARMv8p1);
```

## Library pseudocode for shared/functions/extension/HaveAtomicExtHaveBF16Ext

```
// HaveAtomicExt()
// =====
// HaveBF16Ext()
// =====
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAtomicExt()
    returnHaveBF16Ext()
    return ( HasArchVersion() ||
              (HasArchVersion(ARMv8p2ARMv8p1ARMv8p6);) &&
              boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension"));
```

## Library pseudocode for shared/functions/extension/HaveBF16ExtHaveBTIExt

```
// HaveBF16Ext()
// =====
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.
// HaveBTIExt()
// =====
// Returns TRUE if support for Branch Target Identification is implemented.

boolean HaveBF16Ext()
    return (HaveBTIExt()
    returnHasArchVersion(ARMv8p6ARMv8p5) ||
           (HasArchVersion(ARMv8p2) &&
           boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension"));
```

### Library pseudocode for shared/functions/extension/**HaveBTIExt****HaveBlockBBM**

```
// HaveBTIExt()
// =====
// Returns TRUE if support for Branch Target Identification is implemented.
// HaveBlockBBM()
// =====
// Returns TRUE if support for changing block size without requiring
// break-before-make is implemented.

boolean HaveBTIExt()
HaveBlockBBM()
    return HasArchVersion(ARMv8p5ARMv8p4);
```

### Library pseudocode for shared/functions/extension/**HaveBlockBBM****HaveCNTSCEExt**

```
// HaveBlockBBM()
// HaveCNTSCEExt()
// =====
// Returns TRUE if support for changing block size without requiring
// break-before-make is implemented.
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.

boolean HaveBlockBBM()
    return HaveCNTSCEExt()
    return ( HasArchVersion(ARMv8p4); ) &&
        boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support";
```

### Library pseudocode for shared/functions/extension/**HaveCNTSCEExt****HaveCommonNotPrivateTransExt**

```
// HaveCNTSCEExt()
// =====
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.
// HaveCommonNotPrivateTransExt()
// =====

boolean HaveCNTSCEExt()
    return (HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p4ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support");
```

### Library pseudocode for shared/functions/extension/**HaveCommonNotPrivateTransExt****HaveDGHEExt**

```
// HaveCommonNotPrivateTransExt()
// =====
// HaveDGHEExt()
// =====
// Returns TRUE if Data Gathering Hint instruction support is implemented, and
// FALSE otherwise.

boolean HaveCommonNotPrivateTransExt()
    return HaveDGHEExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension"; HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/**HaveDGHExt****HaveDITExt**

```
// HaveDGHExt()  
// HaveDITExt()  
// =====  
// Returns TRUE if Data Gathering Hint instruction support is implemented, and  
// FALSE otherwise.  
  
boolean HaveDGHExt()  
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";  
    return HaveDITExt();  
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/**HaveDITExt****HaveDOTPExt**

```
// HaveDITExt()  
// =====  
// HaveDOTPExt()  
// =====  
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.  
  
boolean HaveDITExt()  
    return HaveDOTPExt()  
    return ( HasArchVersion(ARMv8p4); ) ||  
            ( HasArchVersion(ARMv8p2) &&  
              boolean IMPLEMENTATION_DEFINED "Has Dot Product extension" );
```

### Library pseudocode for shared/functions/extension/**HaveDOTPExt****HaveDoPD**

```
// HaveDOTPExt()  
// =====  
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.  
// HaveDoPD()  
// =====  
// Returns TRUE if Debug Over Power Down extension  
// support is implemented and FALSE otherwise.  
  
boolean HaveDOTPExt()  
    return ( HaveDoPD()  
    return HasArchVersion(ARMv8p4) ||  
            ( HasArchVersion(ARMv8p2) &&  
              boolean IMPLEMENTATION_DEFINED "Has Dot Product extension" ); ) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension";
```

### Library pseudocode for shared/functions/extension/**HaveDirtyBitModifierExt****HaveDoubleFaultExt**

```
// HaveDirtyBitModifierExt()  
// =====  
// HaveDoubleFaultExt()  
// =====  
  
boolean HaveDirtyBitModifierExt()  
    return HaveDoubleFaultExt()  
    return ( HasArchVersion( ) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESBARMv8p1ARMv8p4 );
```

### Library pseudocode for shared/functions/extension/**HaveDoPD****HaveDoubleLock**

```
// HaveDoPD()  
// =====  
// Returns TRUE if Debug Over Power Down extension  
// support is implemented and FALSE otherwise.  
// HaveDoubleLock()  
// =====  
// Returns TRUE if support for the OS Double Lock is implemented.  
  
boolean HaveDoPD()  
    return HaveDoubleLock()  
    return ( ! HasArchVersion(ARMv8p2ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension"; ) ||  
            boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";
```

### Library pseudocode for shared/functions/extension/~~HaveDoubleFaultExt~~~~HaveE0PDExt~~

```
// HaveDoubleFaultExt()
// =====
// HaveE0PDExt()
// =====
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveDoubleFaultExt()
return (HaveE0PDExt())
return HasArchVersion(ARMv8p4ARMv8p5) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB();};
```

### Library pseudocode for shared/functions/extension/~~HaveDoubleLock~~~~HaveECVExt~~

```
// HaveDoubleLock()
// =====
// Returns TRUE if support for the OS Double Lock is implemented.
// HaveECVExt()
// =====
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.

boolean HaveDoubleLock()
return (!HaveECVExt())
return HasArchVersion(ARMv8p4ARMv8p6) ||
boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";};
```

### Library pseudocode for shared/functions/extension/~~HaveE0PDExt~~~~HaveEMPAMExt~~

```
// HaveE0PDExt()
// =====
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.
// HaveEMPAMExt()
// =====
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.

boolean HaveE0PDExt()
return HaveEMPAMExt()
return ( HasArchVersion() &&
HaveEMPAMExt(ARMv8p5ARMv8p6); ) &&
boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension";};
```

### Library pseudocode for shared/functions/extension/~~HaveECVExt~~~~HaveExtendedCacheSets~~

```
// HaveECVExt()
// =====
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.
// HaveExtendedCacheSets()
// =====

boolean HaveECVExt()
HaveExtendedCacheSets()
return HasArchVersion(ARMv8p6ARMv8p3);
```

## Library pseudocode for shared/functions/extension/HaveEMPAMExtHaveExtendedECDebugEvents

```
// HaveEMPAMExt()  
// =====  
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.  
// HaveExtendedECDebugEvents()  
// =====  
  
boolean HaveEMPAMExt()  
    return (HaveExtendedECDebugEvents()  
    return HasArchVersion(ARMv8p6ARMv8p2) &&  
        HaveEMPAMExt() &&  
        boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension");};
```

## Library pseudocode for shared/functions/extension/HaveExtendedCacheSetsHaveExtendedExecuteNeverExt

```
// HaveExtendedCacheSets()  
// =====  
// HaveExtendedExecuteNeverExt()  
// =====  
  
boolean HaveExtendedCacheSets()  
HaveExtendedExecuteNeverExt()  
    return HasArchVersion(ARMv8p3ARMv8p2);
```

## Library pseudocode for shared/functions/extension/HaveExtendedECDebugEventsHaveFCADDExt

```
// HaveExtendedECDebugEvents()  
// =====  
// HaveFCADDExt()  
// =====  
  
boolean HaveExtendedECDebugEvents()  
HaveFCADDExt()  
    return HasArchVersion(ARMv8p2ARMv8p3);
```

## Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExtHaveFGTEExt

```
// HaveExtendedExecuteNeverExt()  
// =====  
// HaveFGTEExt()  
// =====  
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.  
  
boolean HaveExtendedExecuteNeverExt()  
HaveFGTEExt()  
    return HasArchVersion(ARMv8p2ARMv8p6);
```

## Library pseudocode for shared/functions/extension/HaveFCADDExtHaveFJCVTZSExt

```
// HaveFCADDExt()  
// =====  
// HaveFJCVTZSExt()  
// =====  
  
boolean HaveFCADDExt()  
HaveFJCVTZSExt()  
    return HasArchVersion(ARMv8p3);
```

## Library pseudocode for shared/functions/ extension/~~HaveFGTExt~~~~HaveFP16MulNoRoundingToFP32Ext~~

```
// HaveFGTExt()
// =====
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.
// HaveFP16MulNoRoundingToFP32Ext()
// =====
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise

boolean HaveFGTExt()
returnHaveFP16MulNoRoundingToFP32Ext()
if ! HaveFP16Ext() then return FALSE;
if HasArchVersion() then return TRUE;
return (HasArchVersion(ARMv8p2ARMv8p6ARMv8p4);) &&
boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

## Library pseudocode for shared/functions/extension/~~HaveFJCVTZSExt~~~~HaveFeatCMOW~~

```
// HaveFJCVTZSExt()
// =====
// HaveFeatCMOW()
// =====
// Returns TRUE if the SCTL_R_EL1.CMOW bit is implemented and the SCTL_R_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.

boolean HaveFJCVTZSExt()
HaveFeatCMOW()
return HasArchVersion(ARMv8p3ARMv8p8);
```

## Library pseudocode for shared/functions/ extension/~~HaveFP16MulNoRoundingToFP32Ext~~~~HaveFeatHBC~~

```
// HaveFP16MulNoRoundingToFP32Ext()
// =====
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise
// HaveFeatHBC()
// =====
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.

boolean HaveFP16MulNoRoundingToFP32Ext()
if !HaveFeatHBC()
returnHaveFP16Ext() then return FALSE;
if HasArchVersion(ARMv8p4ARMv8p8) then return TRUE;
return (HasArchVersion(ARMv8p2) &&
boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension"););
```

## Library pseudocode for shared/functions/extension/~~HaveFeatCMOW~~~~HaveFeatHCX~~

```
// HaveFeatCMOW()
// =====
// Returns TRUE if the SCTL_R_EL1.CMOW bit is implemented and the SCTL_R_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.
// HaveFeatHCX()
// =====
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatCMOW()
HaveFeatHCX()
return HasArchVersion(ARMv8p8ARMv8p7);
```

### Library pseudocode for shared/functions/extension/HaveFeatHBCHaveFeatHPMN0

```
// HaveFeatHBC()
// =====
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.
// HaveFeatHPMN0()
// =====
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHBC()
HaveFeatHPMN0()
    return HasArchVersion(ARMv8p8);) && HavePMUv3() && HaveFGTExt() && HaveEL(EL2);
```

### Library pseudocode for shared/functions/extension/HaveFeatHCXHaveFeatLS64

```
// HaveFeatHCX()
// =====
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.
// HaveFeatLS64()
// =====
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatHCX()
    return HaveFeatLS64()
    return ( HasArchVersion(ARMv8p7);) &&
        boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");
```

### Library pseudocode for shared/functions/extension/HaveFeatHPMN0HaveFeatLS64\_ACCDATA

```
// HaveFeatHPMN0()
// =====
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.
// HaveFeatLS64_ACCDATA()
// =====
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatHPMN0()
    return HaveFeatLS64_ACCDATA()
    return ( HasArchVersion(ARMv8p8ARMv8p7) && HavePMUv3HaveFeatLS64_V() &&() &&
        boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte HaveFGTExtEL0() && HaveEL(EL2);with return
```

### Library pseudocode for shared/functions/extension/HaveFeatLS64HaveFeatLS64\_V

```
// HaveFeatLS64()
// =====
// Returns TRUE if the LD64B, ST64B instructions are
// HaveFeatLS64_V()
// =====
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
HaveFeatLS64_V()
    return (HasArchVersion(ARMv8p7) &&
        boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");) && HaveFeatLS64_V()
        boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte with return instruction support");
```

### Library pseudocode for shared/functions/extension/HaveFeatLS64\_ACCDATAHaveFeatMOPS

```
// HaveFeatLS64_ACCDATA()
// =====
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.
// HaveFeatMOPS()
// =====
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
return (HaveFeatMOPS()
return HasArchVersion(ARMv8p7ARMv8p8) && HaveFeatLS64_V() &&
boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte EL0 with return instruction support");;
```

### Library pseudocode for shared/functions/extension/HaveFeatLS64\_VHaveFeatNMI

```
// HaveFeatLS64_V()
// =====
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.
// HaveFeatNMI()
// =====
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.

boolean HaveFeatLS64_V()
return (HaveFeatNMI()
return HasArchVersion(ARMv8p7ARMv8p8) && HaveFeatLS64_V() &&
boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte with return instruction support");;
```

### Library pseudocode for shared/functions/extension/HaveFeatMOPSHaveFeatRPREs

```
// HaveFeatMOPS()
// =====
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.
// HaveFeatRPREs()
// =====
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatMOPS()
return HaveFeatRPREs()
return ( HasArchVersion() &&
(boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate prec
HaveAltFPARMv8p8ARMv8p7);());;
```

### Library pseudocode for shared/functions/extension/HaveFeatNMIHaveFeatTIDCP1

```
// HaveFeatNMI()
// =====
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.
// HaveFeatTIDCP1()
// =====
// Returns TRUE if the SCTL_R_EL1.TIDCP bit is implemented and the SCTL_R_EL2.TIDCP bit
// is implemented if EL2 is implemented.

boolean HaveFeatNMI()
HaveFeatTIDCP1()
return HasArchVersion(ARMv8p8);
```

### Library pseudocode for shared/functions/extension/HaveFeatRPRESHaveFeatWFXT

```
// HaveFeatRPRES()
// =====
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.
// HaveFeatWFXT()
// =====
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatRPRES()
    return (HaveFeatWFXT()
    return HasArchVersion(ARMv8p7) &&
        (boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision"
        HaveAltFP());
```

### Library pseudocode for shared/functions/extension/HaveFeatTIDCP1HaveFeatWFXT2

```
// HaveFeatTIDCP1()
// =====
// Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
// is implemented if EL2 is implemented.
// HaveFeatWFXT2()
// =====
// Returns TRUE if the register number is reported in the ESR_ELx
// on exceptions to WFIT and WFET.

boolean HaveFeatTIDCP1()
HaveFeatWFXT2()
    return HasArchVersionHaveFeatWFXT(ARMv8p8);() && boolean IMPLEMENTATION_DEFINED "Has feature WFXT2";
```

### Library pseudocode for shared/functions/extension/HaveFeatWFXTHaveFeatXS

```
// HaveFeatWFXT()
// =====
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.
// HaveFeatXS()
// =====
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatWFXT()
HaveFeatXS()
    return HasArchVersion(ARMv8p7);
```

### Library pseudocode for shared/functions/extension/HaveFeatWFXT2HaveFlagFormatExt

```
// HaveFeatWFXT2()
// =====
// Returns TRUE if the register number is reported in the ESR_ELx
// on exceptions to WFIT and WFET.
// HaveFlagFormatExt()
// =====
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFeatWFXT2()
HaveFlagFormatExt()
    return (ARMv8p5HaveFeatWFXTHasArchVersion() && boolean IMPLEMENTATION_DEFINED "Has feature WFXT2");)
```

### Library pseudocode for shared/functions/extension/HaveFeatXSHaveFlagManipulateExt

```
// HaveFeatXS()  
// =====  
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier  
// are supported, and FALSE otherwise.  
// HaveFlagManipulateExt()  
// =====  
// Returns TRUE if flag manipulate instructions are implemented.  
  
boolean HaveFeatXS()  
HaveFlagManipulateExt()  
    return HasArchVersion(ARMv8p7ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveFlagFormatExtHaveFrintExt

```
// HaveFlagFormatExt()  
// =====  
// Returns TRUE if flag format conversion instructions implemented.  
// HaveFrintExt()  
// =====  
// Returns TRUE if FRINT instructions are implemented.  
  
boolean HaveFlagFormatExt()  
HaveFrintExt()  
    return HasArchVersion(ARMv8p5);
```

### Library pseudocode for shared/functions/extension/HaveFlagManipulateExtHaveHPMDExt

```
// HaveFlagManipulateExt()  
// =====  
// Returns TRUE if flag manipulate instructions are implemented.  
// HaveHPMDExt()  
// =====  
  
boolean HaveFlagManipulateExt()  
HaveHPMDExt()  
    return HasArchVersionHavePMUv3p1(ARMv8p4);();
```

### Library pseudocode for shared/functions/extension/HaveFrintExtHaveIDSExt

```
// HaveFrintExt()  
// =====  
// Returns TRUE if FRINT instructions are implemented.  
// HaveIDSExt()  
// =====  
// Returns TRUE if ID register handling feature is implemented.  
  
boolean HaveFrintExt()  
HaveIDSExt()  
    return HasArchVersion(ARMv8p5ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveHPMDExtHaveIESB

```
// HaveHPMDExt()  
// =====  
// HaveIESB()  
// =====  
  
boolean HaveHPMDExt()  
    return HaveIESB()  
    return ( HavePMUv3p1HaveRASExt();() &&  
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

## Library pseudocode for shared/functions/extension/HaveIDSExtHaveInt8MatMulExt

```
// HaveIDSExt()
// =====
// Returns TRUE if ID register handling feature is implemented.
// HaveInt8MatMulExt()
// =====
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveIDSExt()
    returnHaveInt8MatMulExt()
    return ( HasArchVersion() ||
              (HasArchVersion(ARMv8p2ARMv8p4ARMv8p6);) &&
              boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension"));
```

## Library pseudocode for shared/functions/extension/HaveIESBHaveLSE2Ext

```
// HaveIESB()
// =====
// HaveLSE2Ext()
// =====
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveIESB()
    return (HaveLSE2Ext()
    return(ARMv8p4HaveRASExtHasArchVersion() &&
           boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier"));
```

## Library pseudocode for shared/functions/extension/HaveInt8MatMulExtHaveMPAMExt

```
// HaveInt8MatMulExt()
// =====
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.
// HaveMPAMExt()
// =====
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
HaveMPAMExt()
    return (HasArchVersion(ARMv8p6) ||
            (HasArchVersion(ARMv8p2) &&
             boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension")); boolean IM
```

## Library pseudocode for shared/functions/extension/HaveLSE2ExtHaveMTE2Ext

```
// HaveLSE2Ext()
// HaveMTE2Ext()
// =====
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveLSE2Ext()
    returnHaveMTE2Ext()
    if ! HasArchVersion(ARMv8p4ARMv8p5);) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";
```

### Library pseudocode for shared/functions/extension/HaveMPAMExtHaveMTE3Ext

```
// HaveMPAMExt()
// HaveMTE3Ext()
// =====
// Returns TRUE if MPAM is implemented, and FALSE otherwise.
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HaveMTE3Ext()
    return ((HasArchVersion() && HaveMTE2Ext()) || (HasArchVersion(ARMv8p5ARMv8p2ARMv8p7) &&
    boolean IMPLEMENTATION_DEFINED "Has MPAM extension"); boolean IMPLEMENTATION_DEFINED
```

### Library pseudocode for shared/functions/extension/HaveMTE2ExtHaveMTEEExt

```
// HaveMTE2Ext()
// =====
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.
// HaveMTEEExt()
// =====
// Returns TRUE if MTE implemented, and FALSE otherwise.

boolean HaveMTE2Ext()
HaveMTEEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension"; if HaveMTE2Ext() then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

### Library pseudocode for shared/functions/extension/HaveMTE3ExtHaveNV2Ext

```
// HaveMTE3Ext()
// =====
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.
// HaveNV2Ext()
// =====
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveMTE3Ext()
    return ((HaveNV2Ext()
    return (HasArchVersion(ARMv8p7ARMv8p4) && HaveMTE2ExtHaveNVExt()) || (HasArchVersion(ARMv8p5) &&
    boolean IMPLEMENTATION_DEFINED "Has MTE3 extension");{)
    && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization";
```

### Library pseudocode for shared/functions/extension/HaveMTEEExtHaveNVExt

```
// HaveMTEEExt()
// =====
// Returns TRUE if MTE implemented, and FALSE otherwise.
// HaveNVExt()
// =====
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveMTEEExt()
    if !HaveNVExt()
    return (HasArchVersion(ARMv8p5ARMv8p3) then
        return FALSE;
    if HaveMTE2Ext() then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension"); &&
    boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization";
```

**Library pseudocode for shared/functions/  
extension/~~HaveNV2Ext~~~~HaveNoSecurePMUDisableOverride~~**

```
// HaveNV2Ext()  
// =====  
// Returns TRUE if Enhanced Nested Virtualization is implemented.  
// HaveNoSecurePMUDisableOverride()  
// =====  
  
boolean HaveNV2Ext()  
    return (HaveNoSecurePMUDisableOverride()  
    return HasArchVersion(ARMv8p4ARMv8p2) && HaveNVExt()  
        && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");};
```

**Library pseudocode for shared/functions/extension/~~HaveNVExt~~~~HaveNoninvasiveDebugAuth~~**

```
// HaveNVExt()  
// =====  
// Returns TRUE if Nested Virtualization is implemented.  
// HaveNoninvasiveDebugAuth()  
// =====  
// Returns TRUE if the Non-invasive debug controls are implemented.  
  
boolean HaveNVExt()  
    return (HaveNoninvasiveDebugAuth()  
    return !HasArchVersion(ARMv8p3ARMv8p4) &&  
        boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization");};
```

**Library pseudocode for shared/functions/  
extension/~~HaveNoSecurePMUDisableOverride~~~~HavePAN3Ext~~**

```
// HaveNoSecurePMUDisableOverride()  
// =====  
// HavePAN3Ext()  
// =====  
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,  
// and FALSE otherwise.  
  
boolean HaveNoSecurePMUDisableOverride()  
HavePAN3Ext()  
    return HasArchVersion() || (HasArchVersion(ARMv8p1ARMv8p2ARMv8p7);) &&  
        boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
```

**Library pseudocode for shared/functions/extension/~~HaveNoninvasiveDebugAuth~~~~HavePANExt~~**

```
// HaveNoninvasiveDebugAuth()  
// =====  
// Returns TRUE if the Non-invasive debug controls are implemented.  
// HavePANExt()  
// =====  
  
boolean HaveNoninvasiveDebugAuth()  
    return !HavePANExt()  
    return HasArchVersion(ARMv8p4ARMv8p1);
```

### Library pseudocode for shared/functions/extension/**HavePAN3ExtHavePMUv3**

```
// HavePAN3Ext()  
// =====  
// Returns TRUE if SCTL_EL1.EPAN and SCTL_EL2.EPAN support is implemented,  
// and FALSE otherwise.  
// HavePMUv3()  
// =====  
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.  
  
boolean HavePAN3Ext()  
    returnHavePMUv3()  
    return boolean IMPLEMENTATION_DEFINED "Has Performance Monitors extension"; HasArchVersion(ARMv8p7)  
    boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
```

### Library pseudocode for shared/functions/extension/**HavePANExtHavePMUv3TH**

```
// HavePANExt()  
// =====  
// HavePMUv3TH()  
// =====  
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.  
  
boolean HavePANExt()  
    returnHavePMUv3TH()  
    return ( HasArchVersion() && HavePMUv3ARMv8p1ARMv8p8);() &&  
    boolean IMPLEMENTATION_DEFINED "Has PMUv3 threshold extension");
```

### Library pseudocode for shared/functions/extension/**HavePMUv3HavePMUv3p1**

```
// HavePMUv3()  
// =====  
// HavePMUv3p1()  
// =====  
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3()  
    return boolean IMPLEMENTATION_DEFINED "Has Performance Monitors extension";HavePMUv3p1()  
    returnHasArchVersion(ARMv8p1) && HavePMUv3();
```

### Library pseudocode for shared/functions/extension/**HavePMUv3THHavePMUv3p4**

```
// HavePMUv3TH()  
// HavePMUv3p4()  
// =====  
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.  
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3TH()  
    return (HavePMUv3p4()  
    returnHasArchVersion(ARMv8p8ARMv8p4) && HavePMUv3() &&  
    boolean IMPLEMENTATION_DEFINED "Has PMUv3 threshold extension");();
```

### Library pseudocode for shared/functions/extension/**HavePMUv3p1HavePMUv3p5**

```
// HavePMUv3p1()  
// HavePMUv3p5()  
// =====  
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.  
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3p1()  
HavePMUv3p5()  
    return HasArchVersion(ARMv8p1ARMv8p5) && HavePMUv3();
```

### Library pseudocode for shared/functions/extension/**HavePMUv3p4HavePMUv3p7**

```
// HavePMUv3p4()  
// HavePMUv3p7()  
// =====  
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.  
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3p4()  
HavePMUv3p7()  
    return HasArchVersion(ARMv8p4ARMv8p7) && HavePMUv3();
```

### Library pseudocode for shared/functions/ extension/**HavePMUv3p5HavePageBasedHardwareAttributes**

```
// HavePMUv3p5()  
// =====  
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.  
// HavePageBasedHardwareAttributes()  
// =====  
  
boolean HavePMUv3p5()  
HavePageBasedHardwareAttributes()  
    return HasArchVersion(ARMv8p5ARMv8p2) && HavePMUv3();};
```

### Library pseudocode for shared/functions/extension/**HavePMUv3p7HavePrivAExt**

```
// HavePMUv3p7()  
// =====  
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.  
// HavePrivAExt()  
// =====  
  
boolean HavePMUv3p7()  
HavePrivAExt()  
    return HasArchVersion(ARMv8p7ARMv8p2) && HavePMUv3();};
```

### Library pseudocode for shared/functions/ extension/**HavePageBasedHardwareAttributesHaveQRDMLAExt**

```
// HavePageBasedHardwareAttributes()  
// =====  
// HaveQRDMLAExt()  
// =====  
  
boolean HavePageBasedHardwareAttributes()  
HaveQRDMLAExt()  
    return HasArchVersion();  
  
boolean HaveAccessFlagUpdateExt()  
    return HasArchVersion(ARMv8p1);  
  
boolean HaveDirtyBitModifierExt()  
    return HasArchVersion(ARMv8p1ARMv8p2ARMv8p1);
```

### Library pseudocode for shared/functions/extension/**HavePrivAExtHaveRASExt**

```
// HavePrivAExt()  
// =====  
// HaveRASExt()  
// =====  
  
boolean HavePrivAExt()  
    return HaveRASExt();  
    return ( HasArchVersion(ARMv8p2);) ||  
        boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

### Library pseudocode for shared/functions/extension/HaveQRDMLAExtHaveRNG

```
// HaveQRDMLAExt()
// =====
// HaveRNG()
// =====
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveQRDMLAExt()
HaveRNG()
    return HasArchVersion(ARMv8p1ARMv8p5);) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";
```

### Library pseudocode for shared/functions/extension/HaveRASExtHaveSBExt

```
// HaveRASExt()
// =====
// HaveSBExt()
// =====
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveRASExt()
    return (HaveSBExt()
    returnHasArchVersion(ARMv8p2ARMv8p5) ||
    boolean IMPLEMENTATION_DEFINED "Has RAS extension");) || boolean IMPLEMENTATION_DEFINED "Has
```

### Library pseudocode for shared/functions/extension/HaveRNGHaveSSBSExt

```
// HaveRNG()
// =====
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.
// HaveSSBSExt()
// =====
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveRNG()
HaveSSBSExt()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension");) || boolean IMP
```

### Library pseudocode for shared/functions/extension/HaveSBExtHaveSecureEL2Ext

```
// HaveSBExt()
// =====
// Returns TRUE if support for SB is implemented, and FALSE otherwise.
// HaveSecureEL2Ext()
// =====
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSBExt()
HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p5ARMv8p4) || boolean IMPLEMENTATION_DEFINED "Has SB extension"););
```

### Library pseudocode for shared/functions/extension/HaveSSBSExtHaveSecureExtDebugView

```
// HaveSSBSExt()
// =====
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.
// HaveSecureExtDebugView()
// =====
// Returns TRUE if support for Secure and Non-secure views of debug peripherals
// is implemented.

boolean HaveSSBSExt()
HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p5ARMv8p4) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension"););
```

### Library pseudocode for shared/functions/extension/HaveSecureEL2ExtHaveSelfHostedTrace

```
// HaveSecureEL2Ext()
// =====
// Returns TRUE if Secure EL2 is implemented.
// HaveSelfHostedTrace()
// =====

boolean HaveSecureEL2Ext()
HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveSecureExtDebugViewHaveSmallTranslationTblExt

```
// HaveSecureExtDebugView()
// =====
// Returns TRUE if support for Secure and Non-secure views of debug peripherals
// is implemented.
// HaveSmallTranslationTblExt()
// =====
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSecureExtDebugView()
    return HaveSmallTranslationTableExt()
    return ( HasArchVersion(ARMv8p4); ) &&
        boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension";
```

### Library pseudocode for shared/functions/extension/HaveSelfHostedTraceHaveSoftwareLock

```
// HaveSelfHostedTrace()
// =====
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

boolean HaveSelfHostedTrace()
    return HaveSoftwareLock( HasArchVersionComponent(component)
    if() then
        return FALSE;
    if HaveDoPD() && component != Component_CTI then
        return FALSE;
    case component of
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            UnreachableARMv8p4Havev8p4Debug);();
```

### Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExtHaveStage2MemAttrControl

```
// HaveSmallTranslationTblExt()
// =====
// Returns TRUE if Small Translation Table Support is implemented.
// HaveStage2MemAttrControl()
// =====
// Returns TRUE if support for Stage2 control of memory types and cacheability
// attributes is implemented.

boolean HaveSmallTranslationTableExt()
    return (HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4) &&
        boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension");;
```

## Library pseudocode for shared/functions/extension/HaveSoftwareLockHaveStatisticalProfiling

```
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.
// HaveStatisticalProfiling()
// =====
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveSoftwareLock(HaveStatisticalProfiling()
    return ComponentHasArchVersion(component)
    if( Havev8p4DebugARMv8p2() then
        return FALSE;
    if HaveDoPD() && component != Component_CTI then
        return FALSE;
    case component of
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable(););
```

## Library pseudocode for shared/functions/extension/HaveStage2MemAttrControlHaveStatisticalProfilingv1p1

```
// HaveStage2MemAttrControl()
// =====
// Returns TRUE if support for Stage2 control of memory types and cacheability
// attributes is implemented.
// HaveStatisticalProfilingv1p1()
// =====
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStage2MemAttrControl()
    return HaveStatisticalProfilingv1p1()
    return ( HasArchVersion(ARMv8p4ARMv8p3);) &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
```

## Library pseudocode for shared/functions/extension/HaveStatisticalProfilingHaveStatisticalProfilingv1p2

```
// HaveStatisticalProfiling()
// =====
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.
// HaveStatisticalProfilingv1p2()
// =====
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfiling()
    return HaveStatisticalProfilingv1p2()
    return ( HasArchVersion() && HaveStatisticalProfilingARMv8p2ARMv8p7;() &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");
```

### Library pseudocode for shared/functions/extension/**HaveStatisticalProfilingv1p1****HaveTWEDExt**

```
// HaveStatisticalProfilingv1p1()
// =====
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.
// HaveTWEDExt()
// =====
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
    return (HaveTWEDExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";HasArchVersion(ARMv8p3) &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
```

### Library pseudocode for shared/functions/extension/**HaveStatisticalProfilingv1p2****HaveTraceExt**

```
// HaveStatisticalProfilingv1p2()
// =====
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveStatisticalProfilingv1p2()
    return (HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";HasArchVersion(ARMv8p7)
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");
```

### Library pseudocode for shared/functions/ extension/**HaveTWEDExt****HaveTrapLoadStoreMultipleDeviceExt**

```
// HaveTWEDExt()
// =====
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.
// HaveTrapLoadStoreMultipleDeviceExt()
// =====

boolean HaveTWEDExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";HaveTrapLoadStoreMultipleDeviceExt()
    returnHasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/**HaveTraceExt****HaveUAOExt**

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.
// HaveUAOExt()
// =====

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";HaveUAOExt()
    returnHasArchVersion(ARMv8p2);
```

**Library pseudocode for shared/functions/  
extension/HaveTrapLoadStoreMultipleDeviceExtHaveV82Debug**

```
// HaveTrapLoadStoreMultipleDeviceExt()  
// =====  
// HaveV82Debug()  
// =====  
  
boolean HaveTrapLoadStoreMultipleDeviceExt()  
HaveV82Debug()  
    return HasArchVersion(ARMv8p2);
```

**Library pseudocode for shared/functions/extension/HaveUAOExtHaveVirtHostExt**

```
// HaveUAOExt()  
// =====  
// HaveVirtHostExt()  
// =====  
  
boolean HaveUAOExt()  
HaveVirtHostExt()  
    return HasArchVersion(ARMv8p2ARMv8p1);
```

**Library pseudocode for shared/functions/extension/HaveV82DebugHavev8p4Debug**

```
// HaveV82Debug()  
// =====  
// Havev8p4Debug()  
// =====  
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.  
  
boolean HaveV82Debug()  
Havev8p4Debug()  
    return HasArchVersion(ARMv8p2ARMv8p4);
```

**Library pseudocode for shared/functions/extension/HaveVirtHostExtInsertIESBBeforeException**

```
// HaveVirtHostExt()  
// =====  
  
// If SCTL_R_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable  
// SError interrupt must be taken before executing any instructions in the exception handler.  
// However, this can be before the branch to the exception handler is made.  
boolean HaveVirtHostExt()  
    return InsertIESBBeforeException(bits(2) el); HasArchVersion(ARMv8p1);
```

**Library pseudocode for shared/  
functions/~~extension~~externalaborts/Havev8p4DebugHandleExternalAbort**

```
// Havev8p4Debug()
// =====
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault. Havev8p4Debug()
returnHandleExternalAbort( HasArchVersionPhysMemRetStatus(memretstatus, boolean iswrite, memaddrdesc,
AccessDescriptor accdesc)
assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
(!HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity,
Fault_AsyncParity}));

fault = NoFault();
fault.statuscode = memretstatus.statuscode;
fault.write = iswrite;
fault.extflag = memretstatus.extflag;
fault.acctype = memretstatus.acctype;
// It is implementation specific whether external aborts signaled
// in-band synchronously are taken synchronously or asynchronously
if (IsExternalSyncAbort(fault) &&
!IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
size, accdesc)) then
if fault.statuscode == Fault_SyncParity then
fault.statuscode = Fault_AsyncParity;
else
fault.statuscode = Fault_AsyncExternal;

if HaveRASExt() then
fault.errortype = PEErrrorState(memretstatus);
else
fault.errortype = bits(2) UNKNOWN;

if IsExternalSyncAbort(fault) then
if UsingAArch32() then
AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
else
AArch64.Abort(memaddrdesc.vaddress, fault);

else
PendSErrorInterruptARMv8p4AddressDescriptor);(fault);
```

**Library pseudocode for shared/  
functions/~~extension~~externalaborts/Havev8p8DebugHandleExternalReadAbort**

```
// Havev8p8Debug()
// =====
// Returns TRUE if support for the Debugv8p8 feature is implemented and FALSE otherwise.

boolean// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read. Havev8p8Debug()
returnHandleExternalReadAbort( HasArchVersionPhysMemRetStatus(memstatus, memaddrdesc,
integer size, AccessDescriptor accdesc)
iswrite = FALSE;
HandleExternalAbortARMv8p8AddressDescriptor);(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## Library pseudocode for shared/

## functions/~~extensionexternalaborts/~~InsertIESBBeforeExceptionHandleExternalTTWAbort

```
// If SCTL_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.
boolean// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord InsertIESBBeforeException(bits(2) el);HandleExternalTTWAbort(PhysMemRetStatus memretstatus, l
AddressDescriptor memaddrdesc,
AccessDescriptor accdesc, integer size,
FaultRecord input_fault)

output_fault = input_fault;
output_fault.extflag = memretstatus.extflag;
output_fault.statuscode = memretstatus.statuscode;
if (IsExternalSyncAbort(output_fault) &&
    !IsExternalAbortTakenSynchronously(memretstatus, iswrite,
memaddrdesc,
size, accdesc)) then
    if output_fault.statuscode == Fault_SyncParity then
        output_fault.statuscode = Fault_AsyncParity;
    else
        output_fault.statuscode = Fault_AsyncExternal;

// If a synchronous fault is on a translation table walk, then update
// the fault type
if IsExternalSyncAbort(output_fault) then
    if output_fault.statuscode == Fault_SyncParity then
        output_fault.statuscode = Fault_SyncParityOnWalk;
    else
        output_fault.statuscode = Fault_SyncExternalOnWalk;
if HaveRASExt() then
    output_fault.errortype = PEErrortype(memretstatus);
else
    output_fault.errortype = bits(2) UNKNOWN;
if !IsExternalSyncAbort(output_fault) then
    PendSErrorInterrupt(output_fault);
    output_fault.statuscode = Fault_None;
return output_fault;
```

## Library pseudocode for shared/functions/ externalaborts/HandleExternalAbortHandleExternalWriteAbort

```
// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalAbort(HandleExternalWriteAbort(PhysMemRetStatus memretstatus, boolean iswrite, memstatus,
      AddressDescriptor memaddrdesc, integer size,
      AccessDescriptor accdesc)
  assert (memretstatus.statuscode IN { iswrite = TRUE; Fault_SyncExternalHandleExternalAbort, Fault_
    (!HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity,
      Fault_AsyncParity}));

  fault = NoFault();
  fault.statuscode = memretstatus.statuscode;
  fault.write = iswrite;
  fault.extflag = memretstatus.extflag;
  fault.acctype = memretstatus.acctype;
  // It is implementation specific whether External aborts signaled
  // in-band synchronously are taken synchronously or asynchronously
  if (IsExternalSyncAbort(fault) &&
    !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
      size, accdesc)) then
    if fault.statuscode == Fault_SyncParity then
      fault.statuscode = Fault_AsyncParity;
    else
      fault.statuscode = Fault_AsyncExternal;

  if HaveRASExt() then
    fault.errortype = PEErrrorState(memretstatus);
  else
    fault.errortype = bits(2) UNKNOWN;

  if IsExternalSyncAbort(fault) then
    if UsingAArch32() then
      AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
    else
      AArch64.Abort(memaddrdesc.vaddress, fault);

  else
    PendSErrorInterrupt(fault);(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## Library pseudocode for shared/functions/

### externalaborts/HandleExternalReadAbortIsExternalAbortTakenSynchronously

```
// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.
boolean

HandleExternalReadAbort(IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus, memstatus,
boolean iswrite, AddressDescriptor memaddrdesc,
integer size, desc,
integer size, AccessDescriptor accdesc)
iswrite = FALSE; accdesc);
HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbortPEErrorState

```
// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord constant bits(2) HandleExternalTTWAbort(Sync_UC = '10'; // Synchronous Uncontainable
constant bits(2) PhysMemRetStatus memretstatus, boolean iswrite, Sync_UER = '00'; // Synchronous Recoverable
constant bits(2)
                                AddressDescriptor memaddrdesc, Sync_UE0 = '11'; // Synchronous Restartable
constant bits(2)
                                AccessDescriptor accdesc, integer size, ASync_UC = '00'; // ASynchronous Unrecoverable
constant bits(2)
                                FaultRecord input_fault)
    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (ASync_UEU = '01'; // ASynchronous Unrecoverable
constant bits(2) IsExternalSyncAbort(output_fault) &&
        !ASync_UER = '11'; // ASynchronous Recoverable
constant bits(2) IsExternalAbortTakenSynchronously(memretstatus, iswrite,
                                memaddrdesc,
                                size, accdesc)) then
        if output_fault.statuscode == ASync_UE0 = '10'; // ASynchronous Restartable
bits(2) Fault_SyncParity then
        output_fault.statuscode = PEErrorState( Fault_AsyncParity PhysMemRetStatus;
    else
        output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update
    // the fault type
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if HaveRASExt() then
        output_fault.errortype = PEErrorState(memretstatus);
    else
        output_fault.errortype = bits(2) UNKNOWN;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    return output_fault; memstatus);
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbortPendSErrorInterrupt

```
// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write. // Pend the SError.

HandleExternalWriteAbort(PendSErrorInterrupt(PhysMemRetStatus FaultRecord memstatus, AddressDescriptor memaddrdesc,
                                integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc); fault);
```

## Library pseudocode for shared/

functions/~~externalabortsfloat/~~~~IsExternalAbortTakenSynchronouslybfloat/BFAdd~~

```
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.
boolean// BFAdd()
// =====
// Single-precision add following BFloat16 computation behaviors.

bits(32) IsExternalAbortTakenSynchronously(BFAdd(bits(32) op1, bits(32) op2)

    bits(32) result; PhysMemRetStatusFPCRTYPE memstatus,
    boolean iswrite, fpcr = FPCR[];
    (type1, sign1, value1) =
        AddressDescriptorBFUnpack desc,
        integer size, (op1);
    (type2, sign2, value2) =
        (op2);
    if type1 == FPTYPE_QNaN || type2 == FPTYPE_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FPTYPE_Infinity);
        inf2 = (type2 == FPTYPE_Infinity);
        zero1 = (type1 == FPTYPE_Zero);
        zero2 = (type2 == FPTYPE_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elseif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0'); // Positive sign when Round to Odd
            else
                result = BFRoundAccessDescriptorBFUnpack accdesc);(result_value);

    return result;
```

## Library pseudocode for shared/functions/externalabortfloat/PEErrorStatebfloat/BFDotAdd

```
constant bits(2) // BFDotAdd()
// =====
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) Sync_UC = '10'; // Synchronous Uncontainable
constant bits(2) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                           bits(16) op2_a, bits(16) op2_b, Sync_UER = '00'; // Synchronous Recoverable
constant bits(2) fpcr_in Sync_UE0 = '11'; // Synchronous Restartable
constant bits(2) fpcr = fpcr_in;

bits(32) prod;

prod = ASync_UC = '00'; // ASynchronous Uncontainable
constant bits(2) ASync_UEU = '01'; // ASynchronous Unrecoverable
constant bits(2) ASync_UER = '11'; // ASynchronous Recoverable
constant bits(2) ASync_UE0 = '10'; // ASynchronous Restartable
result = ASync_UE0 = '10'; // ASynchronous Restartable

bits(2) PEErrState(PhysMemRetStatus memstatus); (addend, prod);

return result;
```

## Library pseudocode for shared/functions/externalabortfloat/PendSErrorInterruptbfloat/BFMatMulAdd

```
// Pend the SError. // BFMatMulAdd()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N)
PendSErrorInterrupt(BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

assert N == 128;

bits(N) result;
bits(32) sum;

for i = 0 to 1
  for j = 0 to 1
    sum = [addend, 2*i + j, 32];
    for k = 0 to 1
      bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
      bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
      bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
      bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
      sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
      ElemFaultRecordElem fault); [result, 2*i + j, 32] = sum;

return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFAddBFMul

```
// BFAdd()
// BFMul()
// =====
// Single-precision add following BFloat16 computation behaviors.
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)
BFMul(bits(16) op1, bits(16) op2)

    bits(32) result;

    FPType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            if (inf1 && zero2) || (zero1 && inf2) then
                result = FPDefaultNaN(fpcr);
            elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
                elseif inf1 || inf2 then
                    result = FPInfinity('0');
                elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
                    (sign1 EOR sign2);
                elseif zero1 || zero2 then
                    result = FPInfinity('1');
                elseif zero1 && zero2 && sign1 == sign2 then
                    result = FPZero(sign1);
                else
                    result_value = value1 + value2;
                    if result_value == 0.0 then
                        result = FPZero('0'); // Positive sign when Round to Odd
                    else
                        result =(sign1 EOR sign2);
                else
                    result = BFRound(result_value);
                (value1*value2);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFDotAddBFMulAdd

```
// BFDotAdd()
// BFMulAdd()
// =====
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b
// Used by BFMLALB and BFMLALT instructions.

bits(32) bits(N) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                        bits(16) op2_a, bits(16) op2_b, BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr = fpcr_in;

    bits(32) prod;

    prod = boolean altfp = BFAAddHaveAltFP(()) && fpcr.AH == '1'; // When TRUE:
    boolean fpexc = !altfp; // Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode
    return BFMulFPMulAdd(op1_a, op2_a), BFMul(op1_b, op2_b));
    result = BFAAdd(addend, prod);

    return result; (addend, op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/bfloat/BFMatMulAddBFNeg

```
// BFMatMulAdd()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])
// BFNeg()
// =====

bits(N) bits(16) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;

    bits(N) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = BFNeg(bits(16) op)
    return NOT(op<15>) : op<14:0>; Elem[addend, 2*i + j, 32];
        for k = 0 to 1
            bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
            bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
            bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
            bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
            sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
            Elem[result, 2*i + j, 32] = sum;

    return result;
```



```

// BFMul()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.
// BFRound()
// =====
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)
BFRound(real op)

    bits(32) result; assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126; E = 8; F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return

    FPCRTypFPZero fpcr = FPCR[];
    (type1,sign1,value1) =(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = BFUnpackMax(op1);
    (type2,sign2,value2) =((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = BFUnpackRoundDown(op2);
    if type1 == (mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FType_QNaN || type2 == FType_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FType_Infinity);
        inf2 = (type2 == FType_Infinity);
        zero1 = (type1 == FType_Zero);
        zero2 = (type2 == FType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr);
        elsif inf1 || inf2 then
            result = FPinfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);

```

```

(sign); // Overflows generate appropriately-signed Infinity
else
    result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

return result;

```

## Library pseudocode for shared/functions/float/bfloat/BFMulAddBFUnpack

```

// BFMulAdd()
// BFUnpack()
// =====
// Used by BFMLALB and BFMLALT instructions.
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

bits(N) (FType, bit, real) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign = fpval<15>;
        exp = fpval<14:7>;
        frac = fpval<6:0> : FPCRTYPEZEROS fpcr_in(16);
    else // N == 32
        sign = fpval<31>;
        exp = fpval<30:23>;
        frac = fpval<22:0>;
    FPCRTYPEFType fpcr = fpcr_in;
    boolean altfp = ftype;
    real value;
    if HaveAltFPisZero() && fpcr.AH == '1'; // When TRUE:
    boolean fpxc = !altfp; // Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode
    return(exp) then
        ftype = ; value = 0.0; // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            ftype = FType_Infinity; value = 2.0^1000000;
        else // no SNaN for BF16 arithmetic
            ftype = FType_QNaN; value = 0.0;
    else
        ftype = FType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UIntFPMulAddFType_Zero(addend, op1, op2, fpcr, fpxc),

    if sign == '1' then value = -value;

    return (ftype, sign, value);

```

## Library pseudocode for shared/functions/float/bfloat/BFNegFPConvertBF

```
// BFNeg()
// =====
// FPConvertBF()
// =====
// Converts a single-precision OP to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) BFNeg(bits(16) op)
return NOT(op<15>) : op<14:0>; FPConvertBF(bits(32) op, FPCRTYPE fpcr_in, FPRounding rounding_in)

FPCRTYPE fpcr = fpcr_in;
FPRounding rounding = rounding_in;
bits(32) result; // BF16 value in top 16 bits
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp; // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
if altfp then rounding = FPRounding_TIEEVEN; // Use RNE rounding mode

// Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
(ftype,sign,value) = FPUnpack(op, fpcr, fpexc);

if ftype == FTYPE_SNaN || ftype == FTYPE_QNaN then
    if fpcr.DN == '1' then
        result = FPDefaultNaN(fpcr);
    else
        result = FPConvertNaN(op);
    if ftype == FTYPE_SNaN then
        if fpexc then FProcessException(FPEXC_InvalidOp, fpcr);
elseif ftype == FTYPE_Infinity then
    result = FPinfinity(sign);
elseif ftype == FTYPE_Zero then
    result = FPZero(sign);
else
    result = FPRoundCVBF(value, fpcr, rounding, fpexc);

// Returns correctly rounded BF16 value from top 16 bits
return result<31:16>;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRTYPE fpcr)
return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

## Library pseudocode for shared/functions/float/bfloat/BFRoundFPRoundCVBF

```
// BFRound()
// =====
// Converts a real number 'op' into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.
// FPRoundCVBF()
// =====
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) BFRound(real op)

    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126; E = 8; F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPRoundCVBF(real op, FPZeroFPCRTType(sign));

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = fpcr, MaxFPRounding((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = rounding, boolean fpexc;
    boolean isbfloat16 = TRUE;
    return RoundDownFPRoundBase(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign); // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result; (op, fpcr, rounding, isbfloat16, fpexc);
```

## Library pseudocode for shared/functions/float/bfloatfixedtofp/BFUnpackFixedToFP

```
// BFUnpack()
// =====
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)
// FixedToFP()
// =====

(FPType, bit, real)// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign = fpval<15>;
        exp = fpval<14:7>;
        frac = fpval<6:0> : FixedToFP(bits(M) op, integer fbits, boolean unsigned, ZerosFPCRTType(16));
    else // N == 32
        sign = fpval<31>;
        exp = fpval<30:23>;
        frac = fpval<22:0>; fpcr,

    FPTYPEFPRounding fptype;
    real value;
    if rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != IsZeroFPRounding_ODD(exp) then
        fptype =;

    // Correct signed-ness
    int operand = FPTYPE_ZeroInt; value = 0.0; // Fixed Flush to Zero
    elsif(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = IsOnesFPZero(exp) then
            if('0');
        else
            result = IsZeroFPRound(frac) then
                fptype = FPTYPE_Infinity; value = 2.0^1000000;
            else // no SNaN for BF16 arithmetic
                fptype = FPTYPE_QNaN; value = 0.0;
        else
            fptype = FPTYPE_Nonzero;
            value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
    (real_operand, fpcr, rounding);

    if sign == '1' then value = -value;

    return (fptype, sign, value); return result;
```

## Library pseudocode for shared/functions/float/bfloatfpabs/FPCConvertBFFPABs

```
// FPCConvertBF()
// =====
// Converts a single-precision 'op' to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.
// FPABs()
// =====

bits(16)bits(N) FPCConvertBF(bits(32) op, FPABs(bits(N) op)

    assert N IN {16,32,64};
    if ! FPCRTYPEUsingAArch32 fpcr_in,() && FPRoundingHaveAltFP rounding_in>() then
        FPCRTYPE fpcr = fpcr_in; fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) =
            FPRounding rounding = rounding_in;
            bits(32) result; // BF16 value in top 16 bits
            boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
            boolean fpexc = !altfp; // Generate no floating-point exceptions
            if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
            if altfp then rounding = FPRounding_TIEEVEN; // Use RNE rounding mode

            // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
            (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

            if fptype == (op, fpcr, FALSE);
            if fptype IN { FPTYPE_SNaN || fptype ==, FPTYPE_QNaN then
                if fpcr.DN == '1' then
                    result = FPDefaultNaN(fpcr);
                else
                    result = FPCConvertNaN(op);
                if fptype == FPTYPE_SNaN then
                    if fpexc then FPPROCESSException(FPEXC_InvalidOp, fpcr);
            elseif fptype == FPTYPE_Infinity then
                result = FPinfinity(sign);
            elseif fptype == FPTYPE_Zero then
                result = FPZero(sign);
            else
                result = FPRoundCVBF(value, fpcr, rounding, fpexc);

            // Returns correctly rounded BF16 value from top 16 bits
            return result<31:16>;

// FPCConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(16) FPCConvertBF(bits(32) op, FPCRTYPE fpcr)
    return FPCConvertBF(op, fpcr, FPRoundingMode(fpcr));} then
    return op; // When fpcr.AH=1, sign of NaN has no consequence

    return '0' : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/bfloatfpadd/FPRoundCVBF/FPAdd

```
// FPRoundCVBF()
// =====
// Converts a real number 'op' into a BFloat16 value using the supplied
// rounding mode 'rounding'. The 'fpexc' argument controls the generation of
// floating-point exceptions.
// FPAdd()
// =====

bits(32)bits(N) FPRoundCVBF(real op, FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr, fpcr)
boolean fpexc = TRUE; // Generate floating-point exceptions
return FPRoundingFPAdd rounding, boolean fpexc)
boolean isbfloat16 = TRUE;
return(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);

(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

boolean altfmaxfmin = FALSE; // Do not use altfp mode for FMIN, FMAX and variants
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);
if !done then
    inf1 = (type1 == FType_Infinity); inf2 = (type2 == FType_Infinity);
    zero1 = (type1 == FType_Zero); zero2 = (type2 == FType_Zero);
    if inf1 && inf2 && sign1 == NOT(sign2) then
        result = FPDefaultNaN(fpcr);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
        result = FPinfinity('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
        result = FPinfinity('1');
    elsif zero1 && zero2 && sign1 == sign2 then
        result = FPZero(sign1);
    else
        result_value = value1 + value2;
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
            result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(result_sign);
        else
            result = FPRound(result_value, fpcr, rounding, fpexc);

    if fpexc then FPProcessDenormsFPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);(type1, type2, M

return result;
```

## Library pseudocode for shared/functions/float/fixedtofpfpcompare/FixedToFPFPCompare

```
// FixedToFP()
// FPCompare()
// =====

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N)bits(4) FixedToFP(bits(M) op, integer fbits, boolean unsigned,FPCompare(bits(N) op1, bits(N) op2,

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPRoundingFPUnpack rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding !=(op1, fpcr);
    (type2,sign2,value2) = FPRounding_ODDFPUnpack;
    (op2, fpcr);

    // Correct signed-ness
    int_operand = bits(4) result;
    if type1 IN { IntFPTType_SNaN(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result =, FPZeroFPTType_QNaN('0');
    else
        result =} || type2 IN { , FPTType_QNaN} then
        result = '0011';
        if type1 == FPTType_SNaN || type2 == FPTType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';

        FPProcessDenormsFPRoundFPTType_SNaN(real_operand, fpcr, rounding);
    (type1, type2, N, fpcr);

    return result;
```

```

// FPAbs()
// =====
// FPCompareEQ()
// =====

bits(N) boolean FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !FPCompareEQ(bits(N) op1, bits(N) op2, UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op, fpcr, FALSE);
    if fptype IN {(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
        result = FALSE;
        if type1 == FType_SNaN || type2 == FType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);

    FPProcessDenorms} then
        return op; // When fpcr.AH=1, sign of NaN has no consequence
(type1, type2, N, fpcr);

    return '0' : op<N-2:0>; return result;

```

## Library pseudocode for shared/functions/float/fpaddfpcomparege/FPAddFPCompareGE

```
// FPAdd()
// =====
// FPCompareGE()
// =====

bits(N)boolean FPAdd(bits(N) op1, bits(N) op2, FPCompareGE(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (op2, fpcr);

    boolean altfmaxfmin = FALSE; // Do not use altfp mode for FMIN, FMAX and variants
    (done,result) = boolean result;
    if type1 IN { FPProcessNaNsFPTType_SNaN(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);
    if !done then
        inf1 = (type1 ==, FPTType_InfinityFPTType_0NaN); inf2 = (type2 ==) || type2 IN { FPTType_Infinity
        zero1 = (type1 ==, FPTType_ZeroFPTType_0NaN); zero2 = (type2 ==) then
        result = FALSE; FPTType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

        if fpexc then, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2); FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

```

// FCompare()
// =====
// FCompareGT()
// =====

bits(4)boolean FCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FCompareGT(bits(N) op1, bits(N) op2, boolean signal_nans) fcomparegt)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUunpack(op1, fpcr);
    (type2,sign2,value2) = FPUunpack(op2, fpcr);

    bits(4) result;
    boolean result;
    if type1 IN {FPTType_SNaN, FPTType_QNaN} || type2 IN {FPTType_SNaN, FPTType_QNaN} then
        result = '0011';
        if type1 == FPTType_SNaN || type2 == FPTType_SNaN || signal_nans then
            FPPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUunpack()
        if value1 == value2 then
            result = '0110';
        elseif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';
            result = (value1 > value2);

        FPPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

```

// FPCmpareEQ()
// =====
// FPConvert()
// =====

boolean // Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPCmpareEQ(bits(N) op1, bits(N) op2, FPConvert(bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = fpcr, FPUunpackFPRounding(op1, fpcr);
    (type2,sign2,value2) = rounding;

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUunpackFPUunpackCV(op2, fpcr);
    (op, fpcr);

    boolean result;
    if type1 IN { alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN, || fptype == FPType_0NaN } || type2 IN { then
        if alt_hp then
            result = FPType_SNaNFPZero, (sign);
        elsif fpcr.DN == '1' then
            result = FPType_0NaNFPDefaultNaN } then
            result = FALSE;
            if type1 == (fpcr);
        else
            result = FPType_SNaNFPConvertNaN || type2 == (op);
            if fptype == FPType_SNaN then || alt_hp then
                FPPProcessException(FPExc_InvalidOp, fpcr);
        else
            // All non-NaN cases can be evaluated on the values produced by FPUunpack()
            result = (value1 == value2);, fpcr);
        elsif fptype ==
            then
            if alt_hp then
                result = sign:Ones(M-1);
                FPPProcessException(FPExc_InvalidOp, fpcr);
            else
                result = FPIInfinity(sign);
            elsif fptype == FPType_Zero then
                result = FPZero(sign);
            else
                result = FPRoundCV(value, fpcr, rounding);

            FPPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
    return FPConvert(op, fpcr, FPRoundingModeFPPProcessDenormsFPTYPE_Infinity(type1, type2, N, fpcr);

    return result;,(fpcr));

```

## Library pseudocode for shared/functions/

float/fpcomparegefpconvertnan/FPCmpareGEFPConvertNaN

```
// FPCmpareGE()
// =====
// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

boolean bits(M) FPCmpareGE(bits(N) op1, bits(N) op2, FPConvertNaN(bits(N) op)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>: FPCRTTypeZeros fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) =(29);
    when 16 frac = op<8:0>: FPUunpackZeros(op1, fpcr);
    (type2,sign2,value2) =(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign: FPUunpackOnes(op2, fpcr);

    boolean result;
    if type1 IN {(M-52):frac;
        when 32 result = sign: FPUtype_SNaNOnes, (M-23):frac<50:29>;
        when 16 result = sign: FPUtype_QNaNOnes} || type2 IN {FPUtype_SNaN, FPUtype_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUunpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);
    (M-10):frac<50:42>;

    return result;
```

## Library pseudocode for shared/functions/float/fpcomparegtfpctype/FPCmpareGTFPCTYPE

```
// FPCmpareGT()
// =====

boolean type FPCmpareGT(bits(N) op1, bits(N) op2, FPCRTType, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUunpack(op1, fpcr);
    (type2,sign2,value2) = FPUunpack(op2, fpcr);

    boolean result;
    if type1 IN {FPUtype_SNaN, FPUtype_QNaN} || type2 IN {FPUtype_SNaN, FPUtype_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUunpack()
        result = (value1 > value2);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

```

// FPConvert()
// =====
// FPDecodeRM()
// =====

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.
// Decode most common AArch32 floating-point rounding encoding.

bits(M) FPRounding FPConvert(bits(N) op, FPDecodeRM(bits(2) rm) FPCRTYPE fpcr, FPRounding rounding)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) =result;
    case rm of
        when '00' result = FPUntpackCVFPRounding_TIEAWAY(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype ==; // A
        when '01' result = FPTYPE_SNaNFPRounding_TIEEVEN || fptype ==; // N
        when '10' result = FPTYPE_QNaNFPRounding_POSINF then
            if alt_hp then
                result =; // P
            when '11' result = FPZeroFPRounding_NEGINF(sign);
            elsif fpcr.DN == '1' then
                result = FPDefaultNaN(fpcr);
            else
                result = FPConvertNaN(op);
            if fptype == FPTYPE_SNaN || alt_hp then
                FPProcessException(FPExc_InvalidOp,fpcr);
            elsif fptype == FPTYPE_Infinity then
                if alt_hp then
                    result = sign:Ones(M-1);
                    FPProcessException(FPExc_InvalidOp, fpcr);
                else
                    result = FPInfinity(sign);
            elsif fptype == FPTYPE_Zero then
                result = FPZero(sign);
            else
                result = FPRoundCV(value, fpcr, rounding);

    FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRTYPE fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr)); // M

return result;

```

## Library pseudocode for shared/functions/

### float/fpconvertnan/fpdecoderounding/FPConvertNaNFPDecodeRounding

```
// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another
// FPDecodeRounding()
// =====

bits(M) // Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPConvertNaN(bits(N) op)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>; FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return ZerosFPRounding_TIEEVEN(29);
        when 16 frac = op<8:0>; // N
        when '01' return ZerosFPRounding_POSINF(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign; // P
        when '10' return OnesFPRounding_NEGINF(M-52):frac;
        when 32 result = sign; // M
        when '11' return OnesFPRounding_ZERO(M-23):frac<50:29>;
        when 16 result = sign: Ones(M-10):frac<50:42>;

    return result; // Z
```

### Library pseudocode for shared/functions/float/fpcrttypefpdefaultnan/FPCRTTypeFPDefaultNaN

```
type // FPDefaultNaN()
// =====

bits(N) FPCRTType; FPDefaultNaN() FPCRTType fpcr = FPCR[];
    return FPDefaultNaN(fpcr);

bits(N) FPDefaultNaN(FPCRTType fpcr)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

    bits(E) exp = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpdecodermfpdiv/FPDecodeRMFPDiv

```
// FPDecoderM()
// =====
// FPDiv()
// =====

// Decode most common AArch32 floating-point rounding encoding.

FPRoundingbits(N) FPDecodeRM(bits(2) rm)FPDiv(bits(N) op1, bits(N) op2,

    FPRoundingFPCRTType result;
    case rm of
        when '00' result = fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPRounding_TIEAWAYFPUnpack; // A
    when '01' result = (op1, fpcr);
    (type2,sign2,value2) = FPRounding_TIEEVENFPUnpack; // N
    when '10' result = (op2, fpcr);
    (done,result) = FPRounding_POSINFPPProcessNaNs; // P
    when '11' result = (type1, type2, op1, op2, fpcr);

    if !done then
        inf1 = type1 == ;
        inf2 = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr);
            FPPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);

        if !zero2 then
            FPPProcessDenormsFPRounding_NEGINF FPType_Infinity; // M
    (type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpdecoderoundingfpexc/FPDecodeRoundingFPExc

```
// FPDecodeRounding()
// =====

// Decode floating-point rounding mode and common AArch64 encoding.

FPRoundingenumeration FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPExc { FPRounding_TIEEVEN; // N
        when '01' return FPExc_InvalidOp, FPRounding_POSINF; // P
        when '10' return FPExc_DivideByZero, FPRounding_NEGINF; // M
        when '11' return FPExc_Overflow, FPExc_Underflow, FPExc_Inexact, FPRounding_ZERO; // ZFPExc_Inf
```

```

// FPDefaultNaN()
// =====
// FPIInfinity()
// =====

bits(N) FPDefaultNaN()FPIInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp =
        FPCRTType fpcr = FPCR[];
    return FPDefaultNaN(fpcr);

bits(N) FPDefaultNaN(FPCRTType fpcr)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

    bits(E) exp = Ones(E);
    bits(F) frac = '1':——bits(F) frac =Zeros(F-1);
(F);

    return sign : exp : frac;

```



```

// FPDIV()
// FPMAX()
// =====

bits(N) FPDIV(bits(N) op1, bits(N) op2, FPMAX(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = boolean altfp = FPUunpackHaveAltFP(op1, fpcr);
    (type2,sign2,value2) = () && ! FPUunpackUsingAArch32(op2, fpcr);
    (done,result) = () && fpcr.AH == '1';
    return FPPROCESSNaNsFPMAX(type1, type2, op1, op2, fpcr);
(op1, op2, fpcr, altfp);

    if !done then
        inf1 = type1 == // FPMAX()
// =====
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPTYPE_Infinity;
    inf2 = type2 == FPMAX(bits(N) op1, bits(N) op2, FPTYPE_InfinityFPCRTYPE;
    zero1 = type1 == fpcr_in, boolean altfp)

    assert N IN {16,32,64}; FPCRTYPE fpcr = fpcr_in;
    (type1,sign1,value1) = FPUunpack(op1, fpcr);
    (type2,sign2,value2) = FPUunpack(op2, fpcr);

    if (altfp && type1 == FPTYPE_Zero;
        zero2 = type2 == && type2 == FPTYPE_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = &&
            ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0')) then
            return FPCDefaultNaNFPZero(fpcr);(sign2);

    (done,result) =
        FPPROCESSExceptionFPPROCESSNaNs((type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then FPEXC_InvalidOpFPTYPE, fpcr);
        elsif inf1 || zero2 then
            result = fptype;
            bit sign;
            real value;
            if value1 > value2 then
                (fptype,sign,value) = (type1,sign1,value1);
            else
                (fptype,sign,value) = (type2,sign2,value2);
            if fptype == FPTYPE_Infinity then
                result = FPIInfinity(sign1 EOR sign2);
                if !inf1 then (sign);
            elsif fptype == FPPROCESSExceptionFPTYPE_Zero then
                sign = sign1 AND sign2; // Use most positive sign
                result = FPEXC_DivideByZeroFPZero, fpcr);
            elsif zero1 || inf2 then
                result = (sign);
            else
                // The use of FPRound() covers the case where there is a trapped underflow exception
                // for a denormalized number even though the result is exact.
                rounding = FPZeroFPRoundingMode(sign1 EOR sign2);
            else
(fpcr);
                if altfp then // Denormal output is not flushed to zero
                    fpcr.FZ = '0';
                    fpcr.FZ16 = '0';

                result = FPRound(value1/value2, fpcr);

    if !zero2 then (value, fpcr, rounding, TRUE);
        FPPROCESSDenorms(type1, type2, N, fpcr);

```

```
return result;
```

## Library pseudocode for shared/functions/float/fpexc/fpmaxnormal/FPExcFPMMaxNormal

```
enumeration// FPMMaxNormal()
// =====
bits(N) FPExc {FPMMaxNormal(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp =FPExc_InvalidOp,(E-1):'0';
    frac = FPExc_DivideByZero, FPExc_Overflow,
           FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};(F);

    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpinfinity/fpmaxnum/FPINfinityFPMMaxNum

```
// FPInfinity()
// =====
// FPMMaxNum()
// =====
bits(N) FPInfinity(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp =FPMMaxNum(bits(N) op1_in, bits(N) op2_in, OnesFPCRTType(E);
    bits(F) frac =fpcr);

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = (op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType_QNaN, FPTType_SNaN};
    boolean type2_nan = type2 IN {FPTType_QNaN, FPTType_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPTType_QNaN && type2 != FPTType_QNaN then
            op1 = FPInfinity('1');
        elsif type1 != FPTType_QNaN && type2 == FPTType_QNaN then
            op2 = FPInfinity('1');

    altfmaxfmin = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMMaxZerosFPUnpack(F);
(op1, op2, fpcr, altfmaxfmin);

    return sign : exp : frac; return result;
```

## Library pseudocode for shared/functions/float/fpmaxfpmerge/FPMaxIsMerging

```
// FPMax()
// =====
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

bits(N) boolean FPMax(bits(N) op1, bits(N) op2, IsMerging( FPCRTType fpcr)
  boolean altfp = boolean merge = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  return() && fpcr.NEP == '1';
  return merge; FPMax(op1, op2, fpcr, altfp);

// FPMax()
// =====
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

  assert N IN {16,32,64};
  FPCRTType fpcr = fpcr_in;
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);

  if (altfp && type1 == FType_Zero && type2 == FType_Zero &&
      ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
    return FPZero(sign2);

  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

  if !done then
    FType fptype;
    bit sign;
    real value;
    if value1 > value2 then
      (fptype,sign,value) = (type1,sign1,value1);
    else
      (fptype,sign,value) = (type2,sign2,value2);
    if fptype == FType_Infinity then
      result = FPInfinity(sign);
    elsif fptype == FType_Zero then
      sign = sign1 AND sign2;          // Use most positive sign
      result = FPZero(sign);
    else
      // The use of FPRound() covers the case where there is a trapped underflow exception
      // for a denormalized number even though the result is exact.
      rounding = FPRoundingMode(fpcr);
      if altfp then // Denormal output is not flushed to zero
        fpcr.FZ = '0';
        fpcr.FZ16 = '0';

      result = FPRound(value, fpcr, rounding, TRUE);

  FPProcessDenorms(type1, type2, N, fpcr);

  return result;
```

## Library pseudocode for shared/functions/float/fpmaxnormalfpmin/FPMaxNormalFPMin

```
// FPMaxNormal()
// =====
// FPMIn()
// =====

bits(N) FPMaxNormal(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp =FPMIn(bits(N) op1, bits(N) op2, OnesFPCRTType(E-1):'0';
    frac =fpcr);
    boolean altfp = () && !UsingAArch32() && fpcr.AH == '1';
    return FPMIn(op1, op2, fpcr, altfp);

// FPMIn()
// =====
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMIn(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == FPType_Zero && type2 == FPType_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then
        FPType fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE);

        FPProcessDenormsOnesHaveAltFP(F);
    (type1, type2, N, fpcr);

    return sign : exp : frac; return result;
```

## Library pseudocode for shared/functions/float/fpmaxnumfpminum/FPMaxNumFPMinNum

```
// FPMaxNum()
// FPMinNum()
// =====

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType_QNaN, FPTType_SNaN};
    boolean type2_nan = type2 IN {FPTType_QNaN, FPTType_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTType_QNaN && type2 != FPTType_QNaN then
            op1 = FPinfinity('1');
        ('0');
        elsif type1 != FPTType_QNaN && type2 == FPTType_QNaN then
            op2 = FPinfinity('1');
        ('0');

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMaxFPMin(op1, op2, fpcr, altfmaxfmin);

    return result;
```

## Library pseudocode for shared/functions/float/fpmergefpmul/IsMergingFPMul

```
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.
// FPMul()
// =====

booleanbits(N) IsMerging(FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean merge =
    assert N IN {16,32,64};
    (type1,sign1,value1) = HaveAltFPFPUnpack() && !(op1, fpcr);
    (type2,sign2,value2) = (op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType_Infinity);
        inf2 = (type2 == FPTType_Infinity);
        zero1 = (type1 == FPTType_Zero);
        zero2 = (type2 == FPTType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPinfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);

        FPProcessDenormsUsingAArch32FPUnpack() && fpcr.NEP == '1';
    return merge; (type1, type2, N, fpcr);

    return result;
```



```

// FPMIn()
// =====
// FPMulAdd()
// =====

bits(N) FPMIn(bits(N) op1, bits(N) op2, FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr),
boolean altfp = boolean fpexc = TRUE; // Generate floating-point exceptions
return HaveAltFPFPMulAdd() && !(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) UsingAArch32() && fpcr.AH == '1';
return FPMIn(op1, op2, fpcr, altfp);

// FPMIn()
// =====
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMIn(bits(N) op1, bits(N) op2, FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr_in,
boolean fpexc)

    assert N IN {16,32,64}; // assert N IN {16,32,64};

    (typeA,signA,valueA) =
    FPCRTTypeFPUnpack fpcr = fpcr_in;
    (addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == (op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero && type2 ==);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero &&
    ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
    return);

    (done,result) = FPZeroFPPProcessNaNs3(sign2);
    (typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    (done,result) = if !( FPPProcessNaNsHaveAltFP(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then() && !
    FPTypeUsingAArch32 fptype;
    bit sign;
    real value;() && fpcr.AH == '1') then
    if typeA ==
    FPRoundingFPTYPE_QNaN rounding;
    if value1 < value2 then
    (fptype,sign,value) = (type1,sign1,value1);
    else
    (fptype,sign,value) = (type2,sign2,value2);
    if fptype ==&& ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN(fpcr);
    if fpexc then FPPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
    infA = (typeA == FPType_Infinity then
    result =); zeroA = (typeA == FPType_Zero);

    // Determine sign and type product will have if it does not cause an
    // Invalid Operation.
    signP = sign1 EOR sign2;

```

```

infP = inf1 || inf2;
zeroP = zero1 || zero2;

// Non SNaN-generated Invalid Operation cases are multiplies of zero
// by infinity and additions of opposite-signed infinities.
invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

if invalidop then
    result = FPDefaultNaN(fpcr);
    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    // Other cases involving infinities produce an infinity of the same sign.
elseif (infA && signA == '0') || (infP && signP == '0') then
    result = FPInfinity(sign);
elseif fptype == ('0');
elseif (infA && signA == '1') || (infP && signP == '1') then
    result = FPType_ZeroFPInfinity then
        sign = sign1 OR sign2; // Use most negative sign
('1');

// Cases where the result is exactly zero and its sign is not determined by the
// rounding mode are additions of same-signed zeros.
elseif zeroA && zeroP && signA == signP then
    result = FPZero(sign);
(signA);

// Otherwise calculate numerical result and round it.
else
    // The use of FPRound() covers the case where there is a trapped underflow exception
    // for a denormalized number even though the result is exact.
    rounding = result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRoundingModeFPRounding_NEGINF(fpcr);
    if altfp then // Denormal output is not flushed to zero
        fpcr.FZ = '0';
        fpcr.FZ16 = '0';

    result = then '1' else '0';
    result = FPZero(result_sign);
    else
        result = FPRound(value, fpcr, rounding, TRUE);(result_value, fpcr, rounding, fpexc);

    if !invalidop && fpexc then
        FPProcessDenormsFPProcessDenorms3(type1, type2, N, fpcr);
(typeA, type1, type2, N, fpcr);

return result;

```



```

// FPMinNum()
// =====
// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, fpcr, fpexc);

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = (boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    FPCRTYPE fpcr, boolean fpexc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(op1, fpcr);
    (type2,-,-) = (addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {(op1, fpcr, fpexc);
    (type2,sign2,value2) = FType_QNaNFPUnpack,(op2, fpcr, fpexc);
    inf1 = (type1 == FType_SNaNFPType_Infinity);
    boolean type2_nan = type2 IN {}; zero1 = (type1 == FType_QNaNFPType_Zero,);
    inf2 = (type2 == FType_SNaNFPType_Infinity);
    boolean altfp =); zero2 = (type2 == FType_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 ==() && fpcr.AH == '1') then
            if typeA == FType_QNaN && type2 !=&& ((inf1 && zero2) || (zero1 && inf2)) then
                result = FType_QNaNFPDefaultNaN then
                    op1 = (fpcr);
                    if fpexc then FPinfinityFPProcessException('0');
                    elsif type1 !=( FType_QNaNFPExc_InvalidOp && type2 ==, fpcr);

    if !done then
        infA = (typeA == FType_QNaNFPType_Infinity then
        op2 =); zeroA = (typeA == FType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPinfinity('0');

    altfmaxfmin = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules

```

```

result =      elif (infA && signA == '1') || (infP && signP == '1') then
              result = ('1');

              // Cases where the result is exactly zero and its sign is not determined by the
              // rounding mode are additions of same-signed zeros.
              elif zeroA && zeroP && signA == signP then
                  result = FPZero(signA);

              // Otherwise calculate numerical result and round it.
              else
                  result_value = valueA + (value1 * value2);
                  if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                      result = FPZero(result_sign);
                  else
                      result = FPRound(result_value, fpcr, rounding, fpexc);

              if !invalidop && fpexc then
                  FPProcessDenormFPMInFPInfinity(op1, op2, fpcr, altfmaxfmin);
(typeA, N, fpcr);

return result;

```

## Library pseudocode for shared/functions/float/fpmulfpmuladdh/FPMulFPPProcessNaNs3H

```
// FPMul()
// =====
// FPPProcessNaNs3H()
// =====

bits(N)(boolean, bits(N)) FPMul(bits(N) op1, bits(N) op2, FPPProcessNaNs3H( FType type1, FType type2, FType type3,
bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
FPCRTYPE fpcr)
fpcr, boolean fpexc)

assert N IN {16,32,64};
(type1,sign1,value1) = assert N IN {32,64};

bits(N) result; FPUunpackFType(op1, fpcr);
(type2,sign2,value2) = type_nan;
// When TRUE, use alternative NaN propagation rules.
boolean altfp = FPUunpackHaveAltFP(op2, fpcr);
(done,result) =() && ! FPPProcessNaNsUsingAArch32(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 ==()) && fpcr.AH == '1';
    boolean op1_nan = type1 IN { FType_InfinityFType_SNaN};
    inf2 = (type2 ==, FType_InfinityFType_QNaN);
    zero1 = (type1 ==);
    boolean op2_nan = type2 IN { FType_ZeroFType_SNaN};
    zero2 = (type2 ==, FType_ZeroFType_QNaN);

    if (inf1 && zero2) || (zero1 && inf2) then
        result =;
    boolean op3_nan = type3 IN { FPDefaultNaNFType_SNaN(fpcr);,
FPPProcessExceptionFType_QNaN};
    if altfp then
        if (type1 == FPExc_InvalidOpFType_SNaN, fpcr);
        elsif inf1 || inf2 then
            result = || type2 == FPInfinityFType_SNaN(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = || type3 == FPZeroFType_SNaN(sign1 EOR sign2);
        else
            result =) then
                type_nan = FPRoundFType_SNaN(value1*value2, fpcr);
            else
                type_nan =

;

boolean done;
if altfp && op1_nan && op2_nan && op3_nan then // <n> register NaN selected
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc));
elsif altfp && op2_nan && (op1_nan || op3_nan) then // <n> register NaN selected
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc));
elsif altfp && op3_nan && op1_nan then // <m> register NaN selected
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op3, fpcr, fpexc));
elsif type1 == FType_SNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FType_SNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc));
elsif type3 == FType_SNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc));
elsif type1 == FType_QNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FType_QNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc));
elsif type3 == FType_QNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc));
else
    done = FALSE; result = ZerosFPPProcessDenormsFType_QNaN(type1, type2, N, fpcr);

return result;(); // 'Don't care' result
return (done, result);
```



```

// FPMulAdd()
// =====
// FPMulX()
// =====

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpxc = TRUE; // Generate floating-point exceptions
    return
    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPMulAdd(addend, op1, op2, fpcr, fpxc);

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpxc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
    FPCRType fpcr, boolean fpxc)

    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUntpack(addend, fpcr, fpxc);
    (type1,sign1,value1) = (op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op1, fpcr, fpxc);
    (type2,sign2,value2) = (op2, fpcr);

    (done,result) = FPUntpackFPProcessNaNs(op2, fpcr, fpxc);
    rounding = (type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPRoundingMode(fpcr));
        inf1 = (type1 == FPType_Infinity); zero1 = (type1 ==);
        inf2 = (type2 == FPType_Zero);
        inf2 = (type2 == FPType_Infinity); zero2 = (type2 ==);
        zero1 = (type1 == FPType_Zero);

    (done,result) = zero2 = (type2 == FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr,

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_0NaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr);
            if fpxc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            if (inf1 && zero2) || (zero1 && inf2) then
                result = FPDefaultNaNFPTwo(fpcr);
                if fpxc then (sign1 EOR sign2);
            elsif inf1 || inf2 then
                result = FPProcessException(FPExc_InvalidOp, fpcr);
            // Other cases involving infinities produce an infinity of the same sign.
            elsif (infA && signA == '0') || (infP && signP == '0') then
                result = FPinfinity('0');
            elsif (infA && signA == '1') || (infP && signP == '1') then
                (sign1 EOR sign2);

```

```

elseif zero1 || zero2 then
    result = FPInfinity('1');

    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
elseif zeroA && zeroP && signA == signP then
    result = FPZero(signA);

    // Otherwise calculate numerical result and round it.
(sign1 EOR sign2);
else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == _____ result = FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc);

    if !invalidop && fpexc then (value1*value2, fpcr);
        FPPProcessDenorms3FPPProcessDenorms(typeA, type1, type2, N, fpcr);
(type1, type2, N, fpcr);

return result;

```



```

// FPMulAddH()
// =====
// Calculates addend + op1*op2.
// FPNeg()
// =====

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPNeg(bits(N) op))

    assert N IN {16,32,64};
    if ! FPCRTYPEUsingAArch32 fpcr)
        boolean fpxc = TRUE; // Generate floating-point exceptions
        return() && FPMulAddHHaveAltFP(addend, op1, op2, fpcr, fpxc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N)() then FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    FPCRTYPE fpcr, boolean fpxc)

    assert N == 32;
    rounding = fpcr = FPCR[];
    if fpcr.AH == '1' then
        (fptype, -, -) = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpxc);
    (type1,sign1,value1) = (op, fpcr, FALSE);
    if fptype IN { FPUntpackFPTYPE_SNaN(op1, fpcr, fpxc);
    (type2,sign2,value2) =, FPUnpack(op2, fpcr, fpxc);
    inf1 = (type1 == FPTYPE_Infinity); zero1 = (type1 == FPTYPE_Zero);
    inf2 = (type2 == FPTYPE_Infinity); zero2 = (type2 == FPTYPE_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpxc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTYPE_0NaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr);
            if fpxc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE_Infinity); zeroA = (typeA == FPTYPE_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr);
            if fpxc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode

```

```

        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc);

    if !invalidop && fpexc then
        FPProcessDenorm(typeA, N, fpcr);
} then

return result; return op; // When fpcr.AH=1, sign of NaN has no consequence

return NOT(op<N-1>) : op<N-2:0>;

```

## Library pseudocode for shared/functions/

float/fpmuladdhfponepointfive/FPPProcessNaNs3HFPOnePointFive

```
// FPPProcessNaNs3H()
// =====
// FPOnePointFive()
// =====

(boolean, bits(N)) bits(N) FPPProcessNaNs3H(FPOnePointFive(bit sign),

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':FTypeOnes type1, FType type2, FType type3,
        bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
        FPCRTYPE fpcr, boolean fpexc)

    assert N IN {32,64};

    bits(N) result;
    FType type_nan;
    // When TRUE, use alternative NaN propagation rules.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FType_SNaN, FType_QNaN};
    boolean op2_nan = type2 IN {FType_SNaN, FType_QNaN};
    boolean op3_nan = type3 IN {FType_SNaN, FType_QNaN};
    if altfp then
        if (type1 == FType_SNaN || type2 == FType_SNaN || type3 == FType_SNaN) then
            type_nan = FType_SNaN;
        else
            type_nan = FType_QNaN;
    end if

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then // <n> register NaN selected
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc));
    elsif altfp && op2_nan && (op1_nan || op3_nan) then // <n> register NaN selected
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc));
    elsif altfp && op3_nan && op1_nan then // <m> register NaN selected
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op3, fpcr, fpexc));
    elsif type1 == FType_SNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FType_SNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc));
    elsif type3 == FType_SNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc));
    elsif type1 == FType_QNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FType_QNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc));
    elsif type3 == FType_QNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc));
    else
        done = FALSE; result = (E-1);
    end if
    frac = '1': Zeros(); // 'Don't care' result
    return (done, result); (F-1);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/ float/fpmulx/fpprocessdenorms/FPMuXFPPProcessDenorm

```
// FPMuX()
// =====

bits(N)// FPPProcessDenorm()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode. FPMuX(bits(N) op1, bits(N) op2, FPPProcessDenorm( FType fpcr)

assert N IN {16,32,64};
bits(N) result;
boolean done;
(type1,sign1,value1) = (boolean altfp = FPUntpackHaveAltFP(op1, fpcr);
(type2,sign2,value2) = ( ) && ! FPUntpackUsingAArch32(op2, fpcr);

(done,result) = ( ) && fpcr.AH == '1';
if altfp && N != 16 && fptype == FPPProcessNaNsFType_Denormal(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == then FType_InfinityFPPProcessException);
    inf2 = (type2 == ( FType_InfinityFPExc_InputDenorm));
    zero1 = (type1 == FType_Zero);
    zero2 = (type2 == FType_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo(sign1 EOR sign2);
    elsif inf1 || inf2 then
        result = FPinfinity(sign1 EOR sign2);
    elsif zero1 || zero2 then
        result = FPZero(sign1 EOR sign2);
    else
        result = FPRound(value1*value2, fpcr);

    FPPProcessDenorms(type1, type2, N, fpcr);

return result; , fpcr);
```

## Library pseudocode for shared/functions/float/fpneg/fpprocessdenorms/FPNegFPPProcessDenorms

```
// FPNeg()
// =====

bits(N)// FPPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode. FPNeg(bits(N) op)

assert N IN {16,32,64};
if !FPPProcessDenorms(UsingAArch32FType() && type1, HaveAltFPFType() then type2, integer N,
    FPCRTYPE fpcr = FPCR[];
    if fpcr.AH == '1' then
        (fptype, -, -) = fpcr;
    boolean altfp = FPUntpackHaveAltFP(op, fpcr, FALSE);
    if fptype IN { ( ) && !FType_SNaNUsingAArch32, ( ) && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == || type2 == FType_Denormal) then
        FPPProcessException(FPExc_InputDenormFType_QNaNFType_Denormal} then

        return op; // When fpcr.AH=1, sign of NaN has no consequence

return NOT(op<N-1>) : op<N-2:0>; , fpcr);
```

## Library pseudocode for shared/functions/

### float/fponepointfivefpprocessdenorms/FPOnePointFiveFPPProcessDenorms3

```
// FPOnePointFive()
// =====

bits(N)// FPPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode. FPOnePointFive(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0';FPPProcessDenorms3(OnesFType(E-1);
    frac = '1':type1, type2, FType type3, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FType_Denormal || type2 == FType_Denormal ||
        type3 == FType_Denormal) then
        FPPProcessException(FPExc_InputDenormZerosFType(F-1);
    result = sign : exp : frac;

    return result; , fpcr);
```

## Library pseudocode for shared/functions/float/

### fpprocessdenorms/FPProcessDenormFPPProcessDenorms4

```
// FPPProcessDenorm()
// =====
// FPPProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorm(FPPProcessDenorms4(FType ftype, integer N, type1, FType type2, FType type3, FType type4,
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FType_Denormal || type2 == FType_Denormal ||
        type3 == FType_Denormal() && fpcr.AH == '1';
    if altfp && N != 16 && ftype == || type4 == FType_Denormal then) then
        FPPProcessException(FPExc_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/

### float/fprocessdenormsfprocessexception/FPPProcessDenormsFPPProcessException

```
// FPPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode. // FPPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPPProcessDenorms(FPPProcessException(FPType FPExc type1, exception, FPType type2, integer N, FPCRTYPE fpcr)
    boolean altfp =
    integer cumul;
    // Determine the cumulative exception bit number
    case exception of
        when HaveAltFPFPExc_InvalidOp() && !cumul = 0;
        when FPExc_DivideByZero cumul = 1;
        when FPExc_Overflow cumul = 2;
        when FPExc_Underflow cumul = 3;
        when FPExc_Inexact cumul = 4;
        when FPExc_InputDenorm cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        bits(8) accumulated_exceptions = GetAccumulatedFPEExceptions();
        accumulated_exceptions<cumul> = '1';
        if boolean IMPLEMENTATION_DEFINED "Process floating-point exception" then
            if UsingAArch32() && fpcr.AH == '1';
        if altfp && N != 16 && (type1 == ()) then FPType_DenormalAArch32.FPTrappedException || type2 == (accumul
            else
                is_ase = FPType_DenormalIsASEInstruction) then();
                FPPProcessExceptionAArch64.FPTrappedException((is_ase, accumulated_exceptions);
            else
                // The exceptions generated by this instruction are accumulated by the PE and
                // FPTrappedException is called later during its execution, before the next
                // instruction is executed. This field is cleared at the start of each FP instruction.(accumul
    elsif UsingAArch32FPExc_InputDenormSetAccumulatedFPEExceptions, fpcr);() then
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';

    return;
```

## Library pseudocode for shared/functions/

float/fpprocessdenormsfpprocessnan/FPPProcessDenorms3FPPProcessNaN

```
// FPPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode. // FPPProcessNaN()
// =====

bits(N)

FPPProcessDenorms3(FPPProcessNaN(FPType type1, fptype, bits(N) op, FPTypeFPCRTYPE type2, fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPPProcessNaN(fptype, op, fpcr, fpexc);

// FPPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPPProcessNaN(FPType type3, integer N, fptype, bits(N) op, FPCRTYPE fpcr)
    boolean altfp = fpcr, boolean fpexc)

    assert N IN {16,32,64};
    assert fptype IN { HaveAltFPEFPType_QNaN() && !, UsingAArch32FPType_SNaN() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 ==);
    integer topfrac;

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_DenormalFPType_SNaN || type2 == then
        result<topfrac> = '1';
        if fpexc then FPType_DenormalFPPProcessException ||
        type3 == ( FPType_DenormalFPExc_InvalidOp) then, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result =
        FPPProcessExceptionFPDefaultNaN(FPExc_InputDenorm, fpcr);(fpcr);

    return result;
```

## Library pseudocode for shared/functions/

float/fprocessdenormsfprocessnans/FProcessDenorms4FProcessNaNs

```
// FProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode. // FProcessNaNs()
// =====

(boolean, bits(N))

FProcessDenorms4(FProcessNaNs(FPType type1, FPType type2, type2, bits(N) op1,
                               bits(N) op2, FPCRTYPE fpcr)
    boolean altfmaxfmin = FALSE; // Do not use alfp mode for FMIN, FMAX and variants
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);

// FProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FProcessNaNs(FPType type3, type1, FPType type4, integer N, type2, bits(N) op1, bits(N)
    boolean altfp = fpcr, boolean altfmaxfmin, boolean fpexc)

    assert N IN {16,32,64};
    bit sign2;
    boolean done;
    bits(N) result;
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == boolean op1_nan = type1 IN { FPType_DenormalFPType_SNaN || type2
        type3 == });
    boolean op2_nan = type2 IN { FPType_DenormalFPType_SNaN || type4 ==, FPType_DenormalFPType_QNaN) then
    boolean any_snan = type1 ==
        FPType_SNaN || type2 == FPType_SNaN;
    FPType type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfmaxfmin && (op1_nan || op2_nan) then
        FProcessException(, fpcr);
        done = TRUE; sign2 = op2 <N-1>;
        result = if type2 == FPType_Zero then FPZero(sign2) else op2;
    elsif altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE; result = FProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = ZerosFPExc_InputDenormFPExc_InvalidOp, fpcr);(); // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/

float/fpprocessexceptionfpprocessnans3/FPPProcessExceptionFPPProcessNaNs3

```

// FPPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate. // FPPProcessNaNs3()
// =====

(boolean, bits(N))

FPPProcessException(FPPProcessNaNs3(FPExc_FType exception, type1, FPType type2, FPType type3,
                                     bits(N) op1, bits(N) op2, bits(N) op3,
                                     FPCRTYPE fpcr)

    integer cumul;
    // Determine the cumulative exception bit number
    case exception of
        when boolean fpexc = TRUE; // Generate floating-point exceptions
        return FPExc_InvalidOpFPPProcessNaNs3 cumul = 0;
        when (type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPExc_DivideByZero cumul = 1;
    when FPPProcessNaNs3( FPExc_OverflowFPType cumul = 2;
    when type1, FPExc_UnderflowFPType cumul = 3;
    when type2, FPExc_InexactFPType cumul = 4;
    when type3,
        bits(N) op1, bits(N) op2, bits(N) op3, FPExc_InputDenormFPCRTYPE cumul

    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        bits(8) accumulated_exceptions = fpcr, boolean fpexc;

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN { GetAccumulatedFPExceptionsFPType_SNaN() };
    accumulated_exceptions<cumul> = '1';
    if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
        if, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    boolean altfp = HaveAltFP() && !UsingAArch32() then() && fpcr.AH == '1';
        AArch32.FPTrappedExceptionFPType(accumulated_exceptions);
    else
        is_ase = type_nan;
    if altfp then
        if type1 == IsASEInstructionFPType_SNaN(); || type2 ==
            AArch64.FPTrappedExceptionFPType_SNaN(is_ase, accumulated_exceptions);
    else
        // The exceptions generated by this instruction are accumulated by the PE and
        // FPTrappedException is called later during its execution, before the next
        // instruction is executed. This field is cleared at the start of each FP instruction. || type
        SetAccumulatedFPExceptionsFPType_SNaN(accumulated_exceptions);
    elsif then
        type_nan = ;
    else
        type_nan = FPType_QNaN;

```

```

boolean done;
if altfp && op1_nan && op2_nan && op3_nan then
    // <n> register NaN selected
    done = TRUE; result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
elsif altfp && op2_nan && (op1_nan || op3_nan) then
    // <n> register NaN selected
    done = TRUE; result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
elsif altfp && op3_nan && op1_nan then
    // <m> register NaN selected
    done = TRUE; result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
elsif type1 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
elsif type3 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type3, op3, fpcr, fpexc);
elsif type1 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
elsif type3 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type3, op3, fpcr, fpexc);
else
    done = FALSE; result = ZerosUsingAArch32FPTYPE_SNaN() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    (); // 'Don't care' result

return; return (done, result);

```

Library pseudocode for shared/functions/

float/fpprocessnanfpreceestimate/FPProcessNaNFPRecipEstimate

```

// FPProcessNaN()
// =====
// FPREcipEstimate()
// =====

bits(N) FPProcessNaN(FPREcipEstimate(bits(N) operand, FPTypeFPCRTYPE fptype, bits(N) op, fpcr_in)

    assert N IN {16,32,64}; FPCRTYPE fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    boolean altfp = FPProcessNaNHaveAltFP(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N)() && ! FPProcessNaN() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode = '00';

    (fptype,sign,value) = FPTYPEFPUntpack fptype, bits(N) op, (operand, fpcr, fpexc); FPCRTYPEFPRounding fpcr

    assert N IN {16,32,64};
    assert fptype IN {rounding = FPTYPE_QNaNFPRoundingMode, (fpcr);
    if fptype == FPTYPE_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == || fptype == FPTYPE_SNaNFPTYPE_QNaN then
        result<topfrac> = '1';
        if fpexc then result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPTYPE_Infinity then
        result = FPZero(sign);
    elsif fptype == FPTYPE_Zero then
        result = FPinfinity(sign);
        if fpexc then FPProcessException(FPExc_InvalidOpFPExc_DivideByZero, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = -- elif (
            (N == 16 && (value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
        ) then
            case rounding of
                when FPRounding_TIEEVEN
                    overflow_to_inf = TRUE;
                when FPRounding_POSINF
                    overflow_to_inf = (sign == '0');
                when FPRounding_NEGINF
                    overflow_to_inf = (sign == '1');
                when FPRounding_ZERO
                    overflow_to_inf = FALSE;
            result = if overflow_to_inf then FPinfinity(sign) else FPMAXNormal(sign);
        if fpexc then
            FPProcessException(FPExc_Overflow, fpcr);
            FPProcessException(FPExc_Inexact, fpcr);
        elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))

```

```

    && (
        (N == 16 && Abs(value) >= 2.0^14) ||
        (N == 32 && Abs(value) >= 2.0^126) ||
        (N == 64 && Abs(value) >= 2.0^1022)
    ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == '0' then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

        if !increasedprecision then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('1':fraction<51:41>);

        integer result_exp;
        case N of
            when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
        // fixed-point number in range [0.5 .. 1.0].
        estimate = RecipEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.
        if !increasedprecision then
            fraction = estimate<7:0> : Zeros(44);
        else
            fraction = estimate<11:0> : ZerosFPDefaultNaNAbs(fpcr);
(40);

        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;

```

```
result_exp = 0;

case N of
  when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
  when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
  when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;
```

Library pseudocode for shared/functions/

float/fpprocessnansfpreestimate/FPProcessNaNsRecipEstimate

```

// FPPProcessNaNs()
// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

(boolean, bits(N)) integer FPPProcessNaNs(RecipEstimate(integer a_in, boolean increasedprecision)
integer a = a_in;
integer r;
if !increasedprecision then
    assert 256 <= a && a < 512;
    a = a*2+1; // Round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // Round to nearest
    assert 256 <= r && r < 512;
else
    assert 2048 <= a && a < 4096;
    a = a*2+1; // Round to nearest
    real real_val = Real(2^25)/Real(a);
    r = FPTypeRoundDown type1, FPType type2, bits(N) op1,
        bits(N) op2, FPCRTYPE fpcr)
boolean altfmaxfmin = FALSE; // Do not use alfp mode for FMIN, FMAX and variants
boolean fpexc = TRUE; // Generate floating-point exceptions
return FPPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);

// FPPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPPProcessNaNs(FPTYPE type1, FPTYPE type2, bits(N) op1, bits(N) op2,
    FPCRTYPE fpcr, boolean altfmaxfmin, boolean fpexc)

assert N IN {16,32,64};
bit sign2;
boolean done;
bits(N) result;
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean op1_nan = type1 IN {FPTYPE_SNaN, FPTYPE_QNaN};
boolean op2_nan = type2 IN {FPTYPE_SNaN, FPTYPE_QNaN};
boolean any_snan = type1 == FPTYPE_SNaN || type2 == FPTYPE_SNaN;
FPTYPE type_nan = if any_snan then FPTYPE_SNaN else FPTYPE_QNaN;

if altfmaxfmin && (op1_nan || op2_nan) then
    FPPProcessException(FPExc_InvalidOp, fpcr);
done = TRUE; sign2 = op2<N-1>;
result = if type2 == FPTYPE_Zero then FPZero(sign2) else op2;
elseif altfp && op1_nan && op2_nan then
    // <n> register NaN selected
done = TRUE; result = FPPProcessNaN(type_nan, op1, fpcr, fpexc);
elseif type1 == FPTYPE_SNaN then
done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPTYPE_SNaN then
done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
elseif type1 == FPTYPE_QNaN then
done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPTYPE_QNaN then

```

```

    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
else
    done = FALSE; result = Zeros(); // 'Don't care' result
(real_val);
    real error = real_val - Real(r);
    boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
    if round_up then r = r+1;
    assert 4096 <= r && r < 8192;

return (done, result); return r;

```



```

// FPProcessNaNs3()
// =====
// FPrepX()
// =====

(boolean, bits(N)) FPProcessNaNs3(FPrepX(bits(N) op, FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRTYPE fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return fpcr_in);

assert N IN {16,32,64}; FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPTYPE type1, FPTYPE type2, FPTYPE type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRTYPE fpcr, boolean fpexc)
fpcr = fpcr_in;
integer esize;
case N of
    when 16 esize = 5;
    when 32 esize = 8;
    when 64 esize = 11;

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN {bits(N) result;
    bits(esize) exp;
    bits(esize) max_exp;
    bits(N-(esize+1)) frac = FPTYPE_SNaNZeros, (});

    boolean altfp = FPTYPE_ONaN};
    boolean op2_nan = type2 IN {FPTYPE_SNaN, FPTYPE_ONaN};
    boolean op3_nan = type3 IN {FPTYPE_SNaN, FPTYPE_ONaN};

    boolean altfp = HaveAltFP() && !() && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    (fptype,sign,value) = UsingAArch32FPUnpack() && fpcr.AH == '1';(op, fpcr, fpexc);

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp =
    FPTYPEOnes type_nan;
    if altfp then
        if type1 == (esize) - 1;

    if fptype == FPTYPE_SNaN || type2 == || fptype == FPTYPE_SNaN || type3 == FPTYPE_SNaN then
        type_nan = FPTYPE_SNaN;
    else
        type_nan = FPTYPE_ONaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE; result = then
        result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then

```

```

    // <n> register NaN selected
    done = TRUE; result = (fptype, op, fpcr, fpexc);
else
    if FPPProcessNaNIsZero(type_nan, op2, fpcr, fpexc);
elseif altfp && op3_nan && op1_nan then
    // <m> register NaN selected
    done = TRUE; result = FPPProcessNaN(type_nan, op3, fpcr, fpexc);
elseif type1 == FPType_SNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPType_SNaN then
    done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
elseif type3 == FPType_SNaN then
    done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
elseif type1 == FPType_QNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPType_QNaN then
    done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
elseif type3 == FPType_QNaN then
    done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
else
    done = FALSE; result = Zeros(); // 'Don't care' result
(exp) then // Zero and denormals
    result = sign:max_exp:frac;
else // Infinities and normals
    result = sign:NOT(exp):frac;

return (done, result); return result;

```



```

// FPRecipEstimate()
// =====
// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRecipEstimate(bits(N) operand, FPRound(real_op, FPCRTType fpcr_in)

    assert N IN {16,32,64}; fpcr_in,
    FPRounding rounding)
    FPCRTType fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    boolean altfp = fpcr.AHP == '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return HaveAltFPFPRoundBase() && !(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode = '00';

    (fptype,sign,value) = FPRound(real_op, FPUntpackFPCRTType(operand, fpcr, fpexc); fpcr_in,
    FPRounding rounding = rounding, boolean fpexc) FPRoundingModeFPCRTType(fpcr);
    if fptype == fpcr == fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPTType_SNaNFPRoundBase || fptype == (op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =====

bits(N) FPTType_QNaN then
    result = FPRound(real_op, FPPProcessNaNFPCRTType(fptype, operand, fpcr, fpexc);
elseif fptype == fpcr)
return FPTType_InfinityFPRound then
    result = (op, fpcr, FPZeroFPRoundingMode(sign));
elseif fptype == FPTType_Zero then
    result = FPinfinity(sign);
    if fpexc then FPPProcessException(FPExc_DivideByZero, fpcr);
elseif (
    (N == 16 && Abs(value) < 2.0^-16) ||
    (N == 32 && Abs(value) < 2.0^-128) ||
    (N == 64 && Abs(value) < 2.0^-1024)
    ) then
    case rounding of
        when FPRounding_TIEEVEN
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO
            overflow_to_inf = FALSE;
    result = if overflow_to_inf then FPinfinity(sign) else FPMMaxNormal(sign);

```

```

    if fpexc then
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
        && (
            (N == 16 && Abs(value) >= 2.0^14) ||
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        else
            // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
            // calculate result exponent. Scaled value has copied sign bit,
            // exponent = 1022 = double-precision biased version of -1,
            // fraction = original fraction
            bits(52) fraction;
            integer exp;
            case N of
                when 16
                    fraction = operand<9:0> : Zeros(42);
                    exp = UInt(operand<14:10>);
                when 32
                    fraction = operand<22:0> : Zeros(29);
                    exp = UInt(operand<30:23>);
                when 64
                    fraction = operand<51:0>;
                    exp = UInt(operand<62:52>);

            if exp == 0 then
                if fraction<51> == '0' then
                    exp = -1;
                    fraction = fraction<49:0>:'00';
                else
                    fraction = fraction<50:0>:'0';

            integer scaled;
            boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

            if !increasedprecision then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('1':fraction<51:41>);

            integer result_exp;
            case N of
                when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
                when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
                when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

            // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
            // fixed-point number in range [0.5 .. 1.0].
            estimate = RecipEstimate(scaled, increasedprecision);

            // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
            // fixed-point result in the range [1.0 .. 2.0].
            // Convert to scaled floating point result with copied sign bit,
            // high-order bits from estimate, and exponent calculated above.
            if !increasedprecision then
                fraction = estimate<7:0> : Zeros(44);
            else
                fraction = estimate<11:0> : Zeros(40);

            if result_exp == 0 then

```

```

        fraction = '1' : fraction<51:1>;
    elsif result_exp == -1 then
        fraction = '01' : fraction<51:2>;
        result_exp = 0;
    end if;

    case N of
        when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
        when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
        when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;
    end case;

    return result; (fper));

```

Library pseudocode for shared/functions/  
float/fpreciestimatefpround/RecipEstimateFPRoundBase

```

// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.
// FPRoundBase()
// =====

integer bits(N) RecipEstimate(integer a_in, boolean increasedprecision)
    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1; // Round to nearest
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1; // Round to nearest
        real real_val = Real(2^25)/Real(a);
        r = FPRoundBase(real op, FPCRTYPE fpcr, FPRounding rounding, boolean isbfloat16)
        boolean fpexc = TRUE; // Generate floating-point exceptions
        return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRoundBase()
// =====
// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRTYPE fpcr, FPRounding rounding,
                    boolean isbfloat16, boolean fpexc)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    integer minimum_exp;
    integer F;
    integer E;
    if N == 16 then
        minimum_exp = -14; E = 5; F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126; E = 8; F = 7;
    elsif N == 32 then
        minimum_exp = -126; E = 8; F = 23;
    else // N == 64
        minimum_exp = -1022; E = 11; F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent + 1;

```

```

while mantissa >= 2.0 do
    mantissa = mantissa / 2.0; exponent = exponent + 1;

// When TRUE, detection of underflow occurs after rounding and the test for a
// denormalized number for single and double precision values occurs after rounding.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

// Deal with flush-to-zero before rounding if FPCR.AH != '1'.
if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
    exponent < minimum_exp) then
    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
        FPSR.UFC = '1';
    else
        if fpexc then FPSR.UFC = '1';
    return FPZero(sign);

biased_exp_unconstrained = (exponent - minimum_exp) + 1;
int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
if !altfp && biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then

    case rounding of
        when FPRounding_TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;

if round_up_unconstrained then
    int_mant_unconstrained = int_mant_unconstrained + 1;
    if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
        biased_exp_unconstrained = biased_exp_unconstrained + 1;
        int_mant_unconstrained = int_mant_unconstrained DIV 2;

// Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
    if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero

```

```

        if fpexc then
            FPSR.UFC = '1';
            FPProcessException(FPExc_Inexact, fpcr);
            return FPZero(sign);
        elseif error != 0.0 || fpcr.UFE == '1' then
            if fpexc then FPProcessException(FPExc_Underflow, fpcr);
        else // altfp == FALSE
            case rounding of
                when FPRounding_TIEEVEN
                    round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                    overflow_to_inf = TRUE;
                when FPRounding_POSINF
                    round_up = (error != 0.0 && sign == '0');
                    overflow_to_inf = (sign == '0');
                when FPRounding_NEGINF
                    round_up = (error != 0.0 && sign == '1');
                    overflow_to_inf = (sign == '1');
                when FPRounding_ZERO, FPRounding_ODD
                    round_up = FALSE;
                    overflow_to_inf = FALSE;

            if round_up then
                int_mant = int_mant + 1;
                if int_mant == 2^F then // Rounded up from denormalized to normalized
                    biased_exp = 1;
                if int_mant == 2^(F+1) then // Rounded up to next exponent
                    biased_exp = biased_exp + 1;
                int_mant = int_mant DIV 2;

            // Handle rounding to odd
            if error != 0.0 && rounding == FPRounding_ODD then
                int_mant<0> = '1';

            // Deal with overflow and generate result.
            if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
                if biased_exp >= 2^E - 1 then
                    result = if overflow_to_inf then FPinfinity(sign) else FPMaxNormal(sign);
                    if fpexc then FPProcessException(FPExc_Overflow, fpcr);
                    error = 1.0; // Ensure that an Inexact exception occurs
                else
                    result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
            else // Alternative half precision
                if biased_exp >= 2^E then
                    result = sign : Ones(N-1);
                    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
                    error = 0.0; // Ensure that an Inexact exception does not occur
                else
                    result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

            // Deal with Inexact exception.
            if error != 0.0 then
                if fpexc then FPProcessException(FPExc_Inexact(real_val);
                real error = real_val - Real(r);
                boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
                if round_up then r = r+1;
                assert 4096 <= r && r < 8192;
            , fpcr);

            return r; // return result;

```

## Library pseudocode for shared/functions/float/fprecpxfpround/FPRecpXFPRoundCV

```
// FPRecpX()
// =====
// FPRoundCV()
// =====
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRecpX(bits(N) op, FPRoundCV(real op, FPCRTType fpcr_in)

    assert N IN {16,32,64}; fpcr_in,
    FPRounding rounding)
    FPCRTType fpcr = fpcr_in;
    integer esize;
    case N of
        when 16 esize = 5;
        when 32 esize = 8;
        when 64 esize = 11;

    bits(N) result;
    bits(esize) exp;
    bits(esize) max_exp;
    bits(N-(esize+1)) frac = fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return ZerosFPRoundBase();

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    (fptype,sign,value) = FPUncpack(op, fpcr, fpexc);

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    if fptype == FPTType_SNaN || fptype == FPTType_QNaN then
        result = FPPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;(op, fpcr, rounding, isbfloat16, fpexc);
```

## Library pseudocode for shared/functions/float/fproundfprounding/FPRoundFPRounding

```
// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) enumeration FPRound(real op, FPRounding { FPCRTType fpcr_in, FPRounding_TIEEVEN, FPRounding rounding
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRounding_NEGINF, FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRounding_ZERO, FPRound(real op, FPRounding_TIEAWAY, FPCRTType fpcr_in, FPRounding rounding, boolean
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =====

bits(N) FPRound(real op, FPCRTType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr)); FPRounding_ODD};
```

Library pseudocode for shared/functions/

float/fproundfproundingmode/FPRoundBaseFPRoundingMode

```

// FPRoundBase()
// =====
// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.

bits(N) FPRounding FPRoundBase(real op, FPRoundingMode( FPCRTType fpcr, fpcr)
return FPRoundingFPDecodeRounding rounding, boolean isbfloat16)
boolean fpexc = TRUE; // Generate floating-point exceptions
return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRoundBase()
// =====
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRTType fpcr, FPRounding rounding,
boolean isbfloat16, boolean fpexc)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    integer minimum_exp;
    integer F;
    integer E;
    if N == 16 then
        minimum_exp = -14; E = 5; F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126; E = 8; F = 7;
    elsif N == 32 then
        minimum_exp = -126; E = 8; F = 23;
    else // N == 64
        minimum_exp = -1022; E = 11; F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);

```

```

error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
if !altfp && biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then

    case rounding of
        when FPRounding_TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained = int_mant_unconstrained DIV 2;

// Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
    if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero
        if fpexc then
            FPSR.UFC = '1';
            FPProcessException(FPExc_Inexact, fpcr);
        return FPZero(sign);
    elsif error != 0.0 || fpcr.UFE == '1' then
        if fpexc then FPProcessException(FPExc_Underflow, fpcr);
else // altfp == FALSE
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;

```

```

        overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then // Rounded up from denormalized to normalized
            biased_exp = 1;
            if int_mant == 2^(F+1) then // Rounded up to next exponent
                biased_exp = biased_exp + 1;
                int_mant = int_mant DIV 2;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding_ODD then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        if fpexc then FPPProcessException(FPExc_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
else // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        if fpexc then FPPProcessException(FPExc_InvalidOp, fpcr);
        error = 0.0; // Ensure that an Inexact exception does not occur
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPPProcessException(FPExc_Inexact, fpcr);

return result; (fpcr.RMode);

```



```

// FPRoundCV()
// =====
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.
// FPRoundInt()
// =====

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundCV(real op, FPRoundInt(bits(N) op, FPCRTYPE fpcr_in, fpcr, FPRounding rounding) rounding, boolean exact)
{
    assert rounding !=
        FPCRTYPE FPRounding_ODD; fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpxc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = () && !UsingAArch32() && fpcr.AH == '1';
    fpxc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype, sign, value) = FPUnpack(op, fpcr, fpxc);

    bits(N) result;
    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTYPE_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPTYPE_Zero then
        result = FPZero(sign);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact FPRoundBaseHaveAltFP(op, fpcr, rounding, isbfloat16, fpxc);

    return result;
}

```



```

enumeration// FPRoundIntN()
// =====

bits(N) FPRounding {FPRoundIntN(bits(N)-op,FPRounding_TIEEVEN,fpcr, FPRounding_POSINF,rounding,integer
    assert rounding !=
        FPRounding_NEGINF,;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = FPRounding_ZERO,() && !
        FPRounding_TIEAWAY,() && fpcr.AH == '1';

    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = (op, fpcr, fpexc);

    if fptype IN {FPTType_SNaN, FPTType_QNaN, FPTType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif fptype == FPTType_Zero then
            result = FPZero(sign);
        else
            // Extract integer component.
            int_result = RoundDown(value);
            error = value - Real(int_result);

            // Determine whether supplied rounding mode requires an increment.
            case rounding of
                when FPRounding_TIEEVEN
                    round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
                when FPRounding_POSINF
                    round_up = error != 0.0;
                when FPRounding_NEGINF
                    round_up = FALSE;
                when FPRounding_ZERO
                    round_up = error != 0.0 && int_result < 0;
                when FPRounding_TIEAWAY
                    round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

            if round_up then int_result = int_result + 1;
            overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

            if overflow then
                if N == 32 then
                    exp = 126 + intsize;
                    result = '1':exp<(E-1):0>:Zeros(F);
                else
                    exp = 1022 + intsize;
                    result = '1':exp<(E-1):0>:Zeros(F);
                    FPProcessException(FPExc_InvalidOp, fpcr);
                // This case shouldn't set Inexact.
                error = 0.0;

            else
                // Convert integer value into an equivalent real value.
                real_result = Real(int_result);

                // Re-encode as a floating-point value, result is always exact.

```

```
    if real_result == 0.0 then
        result = FPZero(sign);
    else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);

    // Generate inexact exceptions.
    if error != 0.0 then
        FPProcessException(FPExc_InexactFPRounding_ODD};, fpcr);

    return result;
```

Library pseudocode for shared/functions/

float/fproundermodefprsqrtestimate/FP RoundingModeFPRSqrtEstimate

```

// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.
// FPRSqrtEstimate()
// =====

FPRoundingbits(N) FPRoundingMode(FPRSqrtEstimate(bits(N) operand, FPCRTYPE fpcr)
return fpcr_in;

assert N IN {16,32,64}; fpcr = fpcr_in;

// When using alternative floating-point behaviour, do not generate
// floating-point exceptions and flush denormal input to zero.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

bits(N) result;
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
    result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elsif fptype == FPTYPE_Zero then
    result = FPInfinity(sign);
    if fpexc then FPProcessException(FPEXC_DivideByZero, fpcr);
elsif sign == '1' then
    result = FPDefaultNaN(fpcr);
    if fpexc then FPProcessException(FPEXC_InvalidOp, fpcr);
elsif fptype == FPTYPE_Infinity then
    result = FPZero('0');
else
    // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
    // evenness or oddness of the exponent unchanged, and calculate result exponent.
    // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
    // biased version of -1 or -2, fraction = original fraction extended with zeros.

    bits(52) fraction;
    integer exp;
    case N of
        when 16
            fraction = operand<9:0> : Zeros(42);
            exp = UInt(operand<14:10>);
        when 32
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        when 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

    if exp == 0 then
        while fraction<51> == '0' do
            fraction = fraction<50:0> : '0';
            exp = exp - 1;
        fraction = fraction<50:0> : '0';

    integer scaled;
    boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

    if !increasedprecision then
        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);
    else
        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:41>);
        else
            scaled = UInt('01':fraction<51:42>);

    integer result_exp;

```

```

case N of
  when 16 result_exp = ( 44 - exp) DIV 2;
  when 32 result_exp = ( 380 - exp) DIV 2;
  when 64 result_exp = (3068 - exp) DIV 2;

estimate = RecipSqrtEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit and high-order
// fraction bits, and exponent calculated above.
case N of
  when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
  when 32
    if !increasedprecision then
      result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
    else
      result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
  when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:ZerosFPDecodeRoundingFPCRTType(fpcr);

return result;

```

Library pseudocode for shared/functions/

float/fproundintfprsqrestimate/FPRoundIntRecipSqrtEstimate

```

// FPRoundInt()
// =====
// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range  $0.25 \leq x < 1.0$ .
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

// Round up to nearest integral floating point value using rounding mode in FPCR/FPSR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) integer FPRoundInt(bits(N) op, RecipSqrtEstimate(integer a_in, boolean increasedprecision))
{
    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then // 0.25 .. 0.5
            a = a*2+1; // a in units of 1/512 rounded to nearest
        else // 0.5 .. 1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = (a+1)*2; // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then // 0.25... 0.5
            a = a*2 + 1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else // 0.5...1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = a+1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);

        real_val = Sqrt(real_val); // This number will lie in the range of 32 to 64
        // Round to nearest even for a DP float number
        real_val = real_val * Real(2^47); // The integer is the size of the whole DP mantissa
        int_val = FPCRTYPERoundDown fpcr, (real_val); // Calculate rounding value
        error = real_val - Real(int_val);
        round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
        int_val = FPRounding rounding, boolean exact)

    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUntpack(op, fpcr, fpexc);

    bits(N) result;
    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then

```

```

    result = FPProcessNaN(fptype, op, fpcr);
elseif fptype == FPType_Infinity then
    result = FPInfinity(sign);
elseif fptype == FPType_Zero then
    result = FPZero(sign);
else
    // Extract integer component.
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Convert integer value into an equivalent real value.
    real_result = Real(int_result);

    // Re-encode as a floating-point value, result is always exact.
    if real_result == 0.0 then
        result = FPZero(sign);
    else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);

    // Generate inexact exceptions.
    if error != 0.0 && exact then
        FPProcessException(FPExc_Inexact, fpcr);
    (real_val); // Round that (to nearest even) to give integer
    error = real_val - Real(int_val);
    round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
    if round_up then int_val = int_val+1;

    return result; r = int_val;
    assert 4096 <= r && r < 8192;

    return r;

```



```

// FPRoundIntN()
// =====
// FPSqrt()
// =====

bits(N) FPRoundIntN(bits(N) op, FPSqrt(bits(N) op, FPCRType fpcr, fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
    (op, fpcr);

    if fptype IN { bits(N) result;
    if fptype == FType_SNaN, || fptype == FType_QNaN, then
        result = FType_Infinity FProcessNaN} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:(fptype, op, fpcr);
        elsif fptype == Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
            FProcessException(FPExc_InvalidOp, fpcr);
        elsif fptype == FType_Zero then
            result = FPZero(sign);
        else
            // Extract integer component.
            int_result = elsif fptype == RoundDown FType_Infinity(value);
            error = value - Real(int_result);

            // Determine whether supplied rounding mode requires an increment.
            case rounding of
                when && sign == '0' then
                    result = FPRounding_TIEEVEN FPinfinity
                    round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
                when (sign);
            elsif sign == '1' then
                    result = FPRounding_POSINF FPDefaultNaN
                    round_up = error != 0.0;
                when (fpcr); FPRounding_NEGINF
                    round_up = FALSE;
                when FPRounding_ZERO
                    round_up = error != 0.0 && int_result < 0;
                when FPRounding_TIEAWAY
                    round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

            if round_up then int_result = int_result + 1;
            overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

            if overflow then
                if N == 32 then
                    exp = 126 + intsize;
                    result = '1':exp<(E-1):0>:Zeros(F);
                else
                    exp = 1022 + intsize;

```

```

        result = '1':exp<(E-1):0>:Zeros(F);
        FPPProcessException(FPExc_InvalidOp, fpcr);
        // This case shouldn't set Inexact.
        error = 0.0;

    else
        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = else
result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, (Sqrt(value), fpcr); FPRounding_ZEROFPPProcessDenorm);

        // Generate inexact exceptions.
        if error != 0.0 then
            FPPProcessException(FPExc_Inexact, fpcr);
(fptype, N, fpcr);

    return result;

```



```

// FPRSqrtEstimate()
// =====
// FPSub()
// =====

bits(N) FPRSqrtEstimate(bits(N) operand, FPSub(bits(N) op1, bits(N) op2, FPCRTYPE fpcr_in)
fpcr);

    assert N IN {16,32,64}; assert N IN {16,32,64};
    rounding =
    FPCRTYPE FPRoundingMode fpcr = fpcr_in;
(fpcr);

    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altfp = (type1,sign1,value1) = HaveAltFPEUnpack() && !(op1, fpcr);
    (type2,sign2,value2) = UsingAArch32() && fpcr.AH == '1';
    boolean fpxc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUunpack(operand, fpcr, fpxc);
(op2, fpcr);

    bits(N) result;
    if fptype == (done,result) = FPUType_SNaNFPPProcessNaNs || fptype == (type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPUType_0NaNFPUType_Infinity then
        result =);
        inf2 = (type2 == FPPProcessNaNFPUType_Infinity(fptype, operand, fpcr, fpxc);
    elseif fptype ==);
        zero1 = (type1 == FPUType_Zero then
        result =);
        zero2 = (type2 == FPIInfinityFPUType_Zero(sign);
        if fpxc then);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPPProcessException(FPExc_DivideByZero, fpcr);
        elseif sign == '1' then
            result = FPDefaultNaN(fpcr);
            if fpxc then(fpcr); FPPProcessException(FPExc_InvalidOp, fpcr);
        elseif fptype == elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPUType_InfinityFPIInfinity then
            result = ('0');
            elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPIInfinity('1');
            elseif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero('0');
        else
            // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
            // evenness or oddness of the exponent unchanged, and calculate result exponent.
            // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
            // biased version of -1 or -2, fraction = original fraction extended with zeros.

            bits(52) fraction;
            integer exp;
            case N of
                when 16
                    fraction = operand<9:0> : (sign1);
            else
                result_value = value1 - value2;
                if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                    result_sign = if rounding == ZerosFPRounding_NEGINF(42);
                    exp = then '1' else '0';
                    result = UIntFPZero(operand<14:10>);
                when 32
                    fraction = operand<22:0> : (result_sign);
            else
                result = ZerosFPRound(29);
                exp = (result_value, fpcr, rounding); UIntFPPProcessDenorms(operand<30:23>);
            when 64

```

```

        fraction = operand<51:0>;
        exp = UInt(operand<62:52>);

    if exp == 0 then
        while fraction<51> == '0' do
            fraction = fraction<50:0> : '0';
            exp = exp - 1;
            fraction = fraction<50:0> : '0';

    integer scaled;
    boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

    if !increasedprecision then
        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);
        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

    integer result_exp;
    case N of
        when 16 result_exp = ( 44 - exp) DIV 2;
        when 32 result_exp = ( 380 - exp) DIV 2;
        when 64 result_exp = (3068 - exp) DIV 2;

    estimate = RecipSqrtEstimate(scaled, increasedprecision);

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
    // Convert to scaled floating point result with copied sign bit and high-order
    // fraction bits, and exponent calculated above.
    case N of
        when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
        when 32
            if !increasedprecision then
                result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            else
                result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
        when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
(type1, type2, N, fpcr);

    return result;

```

```

// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range  $0.25 \leq x < 1.0$ .
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.
// FPThree()
// =====

integer bits(N) RecipSqrtEstimate(integer a_in, boolean increasedprecision)
FPThree(bit sign)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then // 0.25 .. 0.5
            a = a*2+1; // a in units of 1/512 rounded to nearest
        else // 0.5 .. 1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = (a+1)*2; // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that  $b < 2^{14} / \sqrt{a}$ 
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then // 0.25... 0.5
            a = a*2 + 1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else // 0.5..1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = a+1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);

        real_val = Sqrt(real_val); // This number will lie in the range of 32 to 64
        // Round to nearest even for a DP float number
        real_val = real_val * Real(2^47); // The integer is the size of the whole DP mantissa
        int_val = assert N IN {16,32,64};
        constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
        constant integer F = N - (E + 1);
        exp = '1'; RoundDownZeros(real_val); // Calculate rounding value
        error = real_val - Real(int_val);
        round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range  $4096 \leq \text{real\_val} < 8192$ 
        int_val = (E-1);
        frac = '1'; RoundDownZeros(real_val); // Round that (to nearest even) to give integer
        error = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val < 0 == '1'));
        if round_up then int_val = int_val+1;
    (F-1);
    result = sign : exp : frac;

    r = int_val;
    assert 4096 <= r && r < 8192;

    return r; return result;

```

## Library pseudocode for shared/functions/float/fpsqrtfptofixed/FPSqrtFPToFixed

```
// FPSqrt()
// =====
// FPToFixed()
// =====

bits(N) // Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPSqrt(bits(N) op, FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRTyp e fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = fpcr, FPRounding rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);
    (op, fpcr, fpexc);

    bits(N) result;
    // If NaN, set cumulative flag or take exception.
    if fptype == FType_SNaN || fptype == FType_0NaN then
        result =then FPProcessNaNFPProcessException(fptype, op, fpcr);
    elsif fptype == ( FType_ZeroFPExc_InvalidOp then
        result =, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = FPZeroRoundDown(sign);
    elsif fptype == (value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FType_InfinityFPRounding_TIEEVEN && sign == '0' then
            result =round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPinfinityFPRounding_POSINF(sign);
    elsif sign == '1' then
        result =round_up = (error != 0.0);
        when FPDefaultNaNFPRounding_NEGINF(fpcr); round_up = FALSE;
        when
            FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = Sat0(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = — elsif error != 0.0 then FPRoundFPProcessException(Sqrt(value), fpcr);(
        FPProcessDenormFPExc_Inexact(fptype, N, fpcr);
    , fpcr);

    return result;
```



```

// FSub()
// =====
// FPToFixedJS()
// =====

bits(N) // Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FSub(bits(N) op1, bits(N) op2, FPToFixedJS(bits(M) op, FPCRType fpcr)
fpcr, boolean Is64)

    assert N IN {16,32,64};
    rounding = assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpxc_idenorm = !( FPRoundingModeHaveAltFP(fpcr);

    (type1,sign1,value1) =() && ! FPUntpackUsingAArch32(op1, fpcr);
    (type2,sign2,value2) =() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUntpack(op2, fpcr);
    (op, fpcr, fpxc_idenorm);

    (done,result) = Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPProcessNaNsFType_SNaN(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == || fptype == FType_InfinityFType_QNaN);
        inf2 = (type2 == then FType_InfinityFPProcessException);
        zero1 = (type1 == { FType_ZeroFPExc_InvalidOp);
        zero2 = (type2 ==, fpcr);
        Z = '0';

    int_result = FType_ZeroRoundDown);
    (value);
    error = value - Real(int_result);

    if inf1 && inf2 && sign1 == sign2 then
        result = // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32* FPCDefaultNaNRoundUp(fpcr);(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*
            RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
        result = Z = '0';
    elsif error != 0.0 then FPinfinityFPProcessException('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
        result = { FPinfinityFPExc_Inexact('1');
    elsif zero1 && zero2 && sign1 == NOT(sign2) then
        result =, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && ! FPCZeroIsZero(sign1);
    else
        result_value = value1 - value2;
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
            result_sign = if rounding == (op<51:0>) then
                Z = '0';

```

```

if fptype == FPType_Infinity then '1' else '0';
    result = FPZero(result_sign);
else
    result = FPRound(result_value, fpcr, rounding);

FPProcessDenorms(type1, type2, N, fpcr);
then result = 0;

return result; return (result<N-1:0>, Z);

```

### Library pseudocode for shared/functions/float/~~fpthree~~~~fp~~~~two~~/FPThreeFPTwo

```

// FPTThree()
// =====
// FPTTwo()
// =====

bits(N) FPTThree(bit sign)
FPTTwo(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = '1': frac =Zeros(F-1);
(F);
    result = sign : exp : frac;

    return result;

```

## Library pseudocode for shared/functions/float/fptofixedfptype/FPToFixedFPTYPE

```
// FPToFixed()
// =====

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) enumeration FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPTYPE { FPCRTYPE fpcr, FPTYPE_Z

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPTYPE_Denormal, FPRounding_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = FPTYPE_Nonzero, HaveAltFP() && !FPTYPE_Infinity, UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPTYPE_QNaN, FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result; FPTYPE_SNaN};
```

## Library pseudocode for shared/functions/float/fptofixedjsfpunpack/FPToFixedJSFPUnpack

```
// FPToFixedJS()
// =====
// FPUnpack()
// =====

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) (FType, bit, real) FPToFixedJS(bits(M) op, FPUnpack(bits(N) fpval, FPCRTYPE fpcr, boolean 1

    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(fpcr_in)HaveAltFPCRTYPE() && !fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = UsingAArch32FPUnpackBase() && fpcr.AH == '1';
    (fpval, fpcr, fpexc);
    return (fp_type, sign, value);

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype, sign, value) = FPUnpack(

// =====
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FType, bit, real) FPUnpack(op, fpcr, fpexc_idenorm);

    Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPUnpack(bits(N) fpval, FType_SNaNFPCRTYPE || fptype == fpcr_in, boolean fpexc) FType_QN
    fpcr.AHP = '0';
    (fp_type, sign, value) =
        FPProcessExceptionFPUnpackBase(FPExc_InvalidOp, fpcr);
    Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';

    if fptype == FType_Infinity then result = 0;

    return (result<N-1:0>, Z);(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```



```

// FPTwo()
// =====
// FPUnpackBase()
// =====

bits(N) (FType, bit, real) FPTwo(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':FPUnpackBase(bits(N) fpval, ZerosFPCRTType(E-1);
    frac = fpcr);
    boolean fpxc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = (fpval, fpcr, fpxc);
    return (fp_type, sign, value);

// FPUnpackBase()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits and 'fpxc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(FType, bit, real) FPUnpackBase(bits(N) fpval, FPCRTType fpcr_in, boolean fpxc)

    assert N IN {16,32,64};

    FPCRTType fpcr = fpcr_in;

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz = altfp && fpcr.FIZ == '1';
    boolean fz = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FType fptype;

    if N == 16 then
        sign = fpval<15>;
        exp16 = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FType_Zero; value = 0.0;
            else
                fptype = FType_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FType_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FType_QNaN else FType_SNaN;
                value = 0.0;
        else
            fptype = FType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then
        sign = fpval<31>;
        exp32 = fpval<30:23>;
        frac32 = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
                fptype = FType_Zero; value = 0.0;

```

```

elseif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
    fptype = FPType_Zero; value = 0.0;
    // Check whether to raise Input Denormal floating-point exception.
    // fpcr.FIZ==1 does not raise Input Denormal exception.
    if fz then
        // Denormalized input flushed to zero
        if fpexc then FPPProcessException(FPExc_InputDenorm, fpcr);
    else
        fptype = FPType_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
elseif IsOnes(exp32) then
    if IsZero(frac32) then
        fptype = FPType_Infinity; value = 2.0^1000000;
    else
        fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
        value = 0.0;
else
    fptype = FPType_Nonzero;
    value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

else // N == 64
    sign = fpval<63>;
    exp64 = fpval<62:52>;
    frac64 = fpval<51:0>;

    if IsZero(exp64) then
        if IsZero(frac64) then
            // Produce zero if value is zero.
            fptype = FPType_Zero; value = 0.0;
        elseif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
            fptype = FPType_Zero; value = 0.0;
            // Check whether to raise Input Denormal floating-point exception.
            // fpcr.FIZ==1 does not raise Input Denormal exception.
            if fz then
                // Denormalized input flushed to zero
                if fpexc then FPPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elseif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPType_Infinity; value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UIntZerosFPUnpackBase(F);
result = sign : exp : frac;
(frac64)) * 2.0^-52);

return result; if sign == '1' then value = -value;

return (fptype, sign, value);

```

## Library pseudocode for shared/functions/float/fptypefpunpack/FTypeFPUnpackCV

```

enumeration// FPUnpackCV()
// =====
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FType, bit, real) FType {FPUnpackCV(bits(N) fpval, FType_Zero, fpcr_in)
    FType_Denormal, fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) =
        FType_Nonzero,
        FType_Infinity,
        FType_QNaN,
        FType_SNaN};(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

```

## Library pseudocode for shared/functions/float/fpunpackfpzero/FUnpackFPZero

```

// FUnpack()
// =====
// FPZero()
// =====

(FType, bit, real)bits(N) FUnpack(bits(N) fpval, FPZero(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = FPCRTTypeZeros fpcr_in)(E);
    frac =
    FPCRTTypeZeros fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = FUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FUnpack()
// =====
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FType, bit, real) FUnpack(bits(N) fpval, FPCRTType fpcr_in, boolean fpexc)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);(F);
    result = sign : exp : frac;

    return result;

```

Library pseudocode for shared/functions/

float/fpunpackvfpexpandimm/FPUnpackBaseVFPEexpandImm

```

// FPUunpackBase()
// VFPEExpandImm()
// =====

(FPType, bit, real)bits(N) FPUunpackBase(bits(N) fpval,VFPEExpandImm(bits(8) imm8)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>): FPCRTTypeReplicate fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) =(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>: FPUunpackBaseZeros(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUunpackBase()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUunpackBase(bits(N) fpval, FPCRTType fpcr_in, boolean fpexc)

    assert N IN {16,32,64};

    FPCRTType fpcr = fpcr_in;

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz = altfp && fpcr.FIZ == '1';
    boolean fz = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 then
        sign = fpval<15>;
        expl6 = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(expl6) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero; value = 0.0;
            else
                fptype = FPType_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(expl6) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(expl6)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then
        sign = fpval<31>;
        exp32 = fpval<30:23>;
        frac32 = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
                fptype = FPType_Zero; value = 0.0;

```

```

        elsif fz || fiz then          // Flush-to-zero if FIZ==1 or AH,FZ==01
            fptype = FPType_Zero; value = 0.0;
            // Check whether to raise Input Denormal floating-point exception.
            // fpcr.FIZ==1 does not raise Input Denormal exception.
            if fz then
                // Denormalized input flushed to zero
                if fpexc then FPPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
    elsif IsOnes(exp32) then
        if IsZero(frac32) then
            fptype = FPType_Infinity; value = 2.0^1000000;
        else
            fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64
        sign = fpval<63>;
        exp64 = fpval<62:52>;
        frac64 = fpval<51:0>;

        if IsZero(exp64) then
            if IsZero(frac64) then
                // Produce zero if value is zero.
                fptype = FPType_Zero; value = 0.0;
            elsif fz || fiz then          // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero; value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPPProcessException(FPExc_InputDenorm, fpcr);
                else
                    fptype = FPType_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
            elsif IsOnes(exp64) then
                if IsZero(frac64) then
                    fptype = FPType_Infinity; value = 2.0^1000000;
                else
                    fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                    value = 0.0;
                else
                    fptype = FPType_Nonzero;
                    value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
(F-4);
        result = sign : exp : frac;

        if sign == '1' then value = -value;

        return (fptype, sign, value); return result;

```

## Library pseudocode for shared/functions/float/integer/fpunpack/FPUnpackCV/AddWithCarry

```
// FPUnpackCV()
// =====
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(FPType, bit, real)(bits(N), bits(4)) FPUnpackCV(bits(N) fpval, AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
integer unsigned_sum = FPCRTTypeUInt fpcr_in)(x) +
FPCRTTypeUInt fpcr = fpcr_in;
fpcr.FZ16 = '0';
boolean fpexc = TRUE; // Generate floating-point exceptions
(fp_type, sign, value) = (y) + (carry_in);
integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
bit n = result<N-1>;
bit z = if IsZero(result) then '1' else '0';
bit c = if UInt(result) == unsigned_sum then '0' else '1';
bit v = if SIntFPUnpackBaseUInt(fpval, fpcr, fpexc);
return (fp_type, sign, value); (result) == signed_sum then '0' else '1';
return (result, n:z:c:v);
```

## Library pseudocode for shared/functions/float/interrupts/fpzero/FPZeroInterruptID

```
// FPZero()
// =====

bits(N) enumeration FPZero(bit sign)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp =InterruptID { Zeros(E);
frac =InterruptID_PMUIRQ, InterruptID_COMMIRQ,
InterruptID_CTIIRQ,
InterruptID_COMMRX,
InterruptID_COMMTX,
InterruptID_CNTP,
InterruptID_CNTHP,
InterruptID_CNTHPS,
InterruptID_CNTPS,
InterruptID_CNTV,
InterruptID_CNTHV,
Zeros(F);
result = sign : exp : frac;

return result; InterruptID_CNTHVS,
};
```

## Library pseudocode for shared/functions/floatinterrupts/vfpexpandimm/ VFPEExpandImmSetInterruptRequestLevel

```
// VFPEExpandImm()
// =====

bits(N) // Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel( VFPEExpandImm(bits(8) imm8)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;id, signal_level);
```

## Library pseudocode for shared/functions/integermemory/AddWithCarryAArch64.BranchAddr

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags
// AArch64.BranchAddr()
// =====
// Return the virtual address with tag bits removed for storing to the program counter.

(bits(N), bits(4))bits(64) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum =AArch64.BranchAddr(bits(64) vaddress)
    assert ! UIntUsingAArch32(x) +();
    msbit = UIntAddrTop(y) +(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN { UIntEL0(carry_in);
        integer signed_sum =, SIntEL1(x) +} || SIntIsInHost(y) +()) && vaddress<msbit> == '1' then
        return UIntSignExtend(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if(vaddress<msbit:0>);
    else
        return IsZeroZeroExtend(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);(vaddress<msbit:0>);
```

## Library pseudocode for shared/functions/interruptsmemory/InterruptIDAccType

```

enumeration InterruptID {AccType {
    InterruptID_PMUIRQ, AccType_NORMAL,           // Normal loads and stores
    InterruptID_COMMIRQ, AccType_STREAM,          // Streaming loads and stores
    InterruptID_CTIIRQ, AccType_VEC,              // Vector loads and stores
    InterruptID_COMMRX, AccType_VECSTREAM,        // Streaming vector loads and stores
    InterruptID_COMMTX, AccType_UNPRIVSTREAM,     // Streaming unprivileged loads and stores
    InterruptID_CNTP, AccType_A32LSMD,            // Load and store multiple
    InterruptID_CNTHP, AccType_ATOMIC,            // Atomic loads and stores
    InterruptID_CNTHPS, AccType_ATOMICRW,         // Load-Acquire and Store-Release
    InterruptID_CNTPS, AccType_ORDERED,           // Load-Acquire and Store-Release
    InterruptID_CNTV, AccType_ORDEREDRW,          // Load-Acquire and Store-Release with atomic access
    InterruptID_CNTHV, AccType_ORDEREDATOMIC,     // Load-Acquire and Store-Release with atomic access
    InterruptID_CNTHVS,
}; AccType_ORDEREDATOMICRW, AccType_ATOMICLS64,   // Atomic 64-byte loads and stores
    AccType_LIMITEDORDERED, // Load-LOAcquire and Store-LORelease
    AccType_UNPRIV,         // Load and store unprivileged
    AccType_IFETCH,         // Instruction fetch
    AccType_TTW,            // Translation table walk
    AccType_NV2REGISTER,    // MRS/MSR instruction used at EL1 and which is
                           // converted to a memory access that uses the
                           // EL2 translation regime
    // Other operations
    AccType_DC,             // Data cache maintenance
    AccType_IC,             // Instruction cache maintenance
    AccType_DCZVA,          // DC ZVA instructions
    AccType_ATPAN,          // Address translation with PAN permission checks
    AccType_AT};           // Address translation

```

## Library pseudocode for shared/functions/interruptsmemory/SetInterruptRequestLevelAccessDescriptor

```

// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(typeAccessDescriptor is {
    MPAMInfo mpam,
    AccTypeInterruptID id, signal level); acctype)

```

## Library pseudocode for shared/functions/memory/AArch64.BranchAddrAddrTop

```

// AArch64.BranchAddr()
// =====
// Return the virtual address with tag bits removed for storing to the program counter.
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

bits(64) integer AArch64.BranchAddr(bits(64) vaddress)
    assert !AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert UsingAArch32HaveEL();
    msbit = {el};
    regime = AddrTopS1TranslationRegime(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {el});
    if EL0ELUsingAArch32, (regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EL1EffectiveTBI || IsInHost() && vaddress<msbit> == '1' then
            return SignExtend(vaddress<msbit:0>);
        else
            return ZeroExtend(vaddress<msbit:0>); {address, IsInstr, el} == '1' then
                return 55;
            else
                return 63;

```

## Library pseudocode for shared/functions/memory/**AccTypeAllocation**

```

enumeration constant bits(2) AccType {MemHint_No = '00'; // No Read-Allocate, No Write-Allocate
constant bits(2) AccType_NORMAL, // Normal loads and stores MemHint_WA = '01'; // No Read-Allocate
constant bits(2) AccType_STREAM, // Streaming loads and stores MemHint_RA = '10'; // Read-Allocate
constant bits(2) AccType_VEC, // Vector loads and stores MemHint_RWA = '11'; // Read-Allocate, Write-Allocate
AccType_VECSTREAM, // Streaming vector loads and stores
AccType_UNPRIVSTREAM, // Streaming unprivileged loads and stores
AccType_A32LSMD, // Load and store multiple
AccType_ATOMIC, // Atomic loads and stores
AccType_ATOMICRW, // Atomic loads and stores with write-back
AccType_ORDERED, // Load-Acquire and Store-Release
AccType_ORDEREDRW, // Load-Acquire and Store-Release with write-back
AccType_ORDEREDATOMIC, // Load-Acquire and Store-Release with atomic access
AccType_ORDEREDATOMICRW, // Load-Acquire and Store-Release with atomic access and write-back
AccType_ATOMICLS64, // Atomic 64-byte loads and stores
AccType_LIMITEDORDERED, // Load-LOAcquire and Store-LORelease
AccType_UNPRIV, // Load and store unprivileged
AccType_IFETCH, // Instruction fetch
AccType_TTW, // Translation table walk
AccType_NV2REGISTER, // MRS/MSR instruction used at EL1 and which is
// converted to a memory access that uses the
// EL2 translation regime
// Other operations
AccType_DC, // Data cache maintenance
AccType_IC, // Instruction cache maintenance
AccType_DCZVA, // DC ZVA instructions
AccType_ATPAN, // Address translation with PAN permission checks
AccType_AT}; // Address translation

```

## Library pseudocode for shared/functions/memory/**AccessDescriptorBigEndian**

```

type// BigEndian()
// =====
boolean AccessDescriptor is (BigEndian(
MPAMInfoAccType mpam, acctype)
boolean bigend;
if
() && acctype == AccType_NV2REGISTER then
return SCTL_EL2.EE == '1';
if UsingAArch32() then
bigend = (PSTATE.E != '0');
elsif PSTATE.EL == EL0 then
bigend = (SCTLR[0].E0E != '0');
else
bigend = (SCTLR[AccTypeHaveNV2Ext acctype][0].EE != '0');
return bigend;

```

## Library pseudocode for shared/functions/memory/AddrTopBigEndianReverse

```
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.
// BigEndianReverse()
// =====

integer bits(width) AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assertBigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return HaveELBigEndianReverse(el);
    regime =(value<half-1:0>) : S1TranslationRegimeBigEndianReverse(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;(value<width-1:half>);
```

## Library pseudocode for shared/functions/memory/AllocationCacheability

```
constant bits(2) MemHint_No = '00'; // No Read-Allocate, No Write-Allocate
MemAttr_NC = '00'; // Non-cacheable
constant bits(2) MemHint_WA = '01'; // No Read-Allocate, Write-Allocate
MemAttr_WT = '10'; // Write-through
constant bits(2) MemHint_RA = '10'; // Read-Allocate, No Write-Allocate
constant bits(2) MemAttr_WB = '11'; // Write-back MemHint_RWA = '11'; // Read-Allocate, Write-Allocate
```

## Library pseudocode for shared/functions/memory/BigEndianCreateAccessDescriptor

```
// BigEndian()
// =====
// CreateAccessDescriptor()
// =====

boolean AccessDescriptor BigEndian(CreateAccessDescriptor(AccType acctype)
    boolean bigend;
    if(acctype) HaveNV2ExtAccessDescriptor() && acctype == accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = AccType_NV2REGISTERGenMPAMcurEL then
        return SCTL_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].EOE != '0');
    else
        bigend = (SCTL_EL2.EE != '0');
    return bigend;(acctype);
    return accdesc;
```

## Library pseudocode for shared/functions/memory/BigEndianReverseDataMemoryBarrier

```
// BigEndianReverse()
// =====

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    returnDataMemoryBarrier( BigEndianReverseMBReqDomain(value<half-1:0>) :domain, BigEndianReverseMBReqDomain(value<half-1:0>));
```

## Library pseudocode for shared/functions/memory/CacheabilityDataSynchronizationBarrier

```
constant bits(2) MemAttr_NC = '00'; // Non-cacheable
constant bits(2) DataSynchronizationBarrier( MemAttr_WT = '10'; // Write-through
constant bits(2) domain, MemAttr_WB = '11'; // Write-back types, boolean nXS);
```

## Library pseudocode for shared/functions/memory/CreateAccessDescriptorDeviceType

```
// CreateAccessDescriptor()
// =====
AccessDescriptor enumeration CreateAccessDescriptor(DeviceType {AccType acctype} DeviceType_GRE,
AccessDescriptor accdesc;
accdesc.acctype = acctype;
accdesc.mpam = DeviceType_nGRE, DeviceType_nGnRE, GenMPAMcurEL(acctype);
return accdesc; DeviceType_nGnRnE};
```

## Library pseudocode for shared/functions/memory/DataMemoryBarrierEffectiveTBI

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit DataMemoryBarrier(EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
bit tbi;
bit tbid;
assert(MBReqDomainHaveEL domain, (el);
regime = (el);
assert(!ELUsingAArch32(regime));

case regime of
when EL1
tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
if HavePACExt() then
tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
when EL2
if HaveVirtHostExt() && ELIsInHost(el) then
tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
if HavePACExt() then
tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
else
tbi = TCR_EL2.TBI;
if HavePACExt() then tbid = TCR_EL2.TBID;
when EL3
tbi = TCR_EL3.TBI;
if HavePACExt() then tbid = TCR_EL3.TBID;

return (if tbi == '1' && (!HavePACExtMBReqTypesS1TranslationRegime types);() || tbid == '0' || !IsIn
```

## Library pseudocode for shared/functions/memory/DataSynchronizationBarrierEffectiveTCMA

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit DataSynchronizationBarrier(EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert(MBReqDomainHaveEL domain,(el);
    regime = (el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3MBReqTypesS1TranslationRegime types, boolean nXS);tcma = TCR_EL3.TCMA;

    return tcma;
```

## Library pseudocode for shared/functions/memory/DeviceTypeFault

```
enumeration DeviceType {Fault {DeviceType_GRE, Fault_None, DeviceType_nGRE, Fault_AccessFlag, DeviceType_nGRE,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_HWUpdateAccessFlag,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint};
```

## Library pseudocode for shared/functions/memory/EffectiveTBIFaultRecord

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bittype EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assertFaultRecord is ( HaveELFault(el);
    regime = statuscode, // Fault Status S1TranslationRegimeAccType(el);
    assert(!acctype, // Type of access that faultedELUsingAArch32FullAddress(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACEExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                    if HavePACEExt() then
                        tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
                else
                    tbi = TCR_EL2.TBI;
                    if HavePACEExt() then tbid = TCR_EL2.TBID;
            when EL3
                tbi = TCR_EL3.TBI;
                if HavePACEExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACEExt() || tbid == '0' || !IsInstr) then '1' else '0');ipaddress, //
        boolean s2fslwalk, // Is on a Stage 1 translation table walk
        boolean write, // TRUE for a write, FALSE for a read
        integer level, // For translation, access flag and permission faults
        bit extflag, // IMPLEMENTATION DEFINED syndrome for External aborts
        boolean secondstage, // Is a Stage 2 abort
        bits(4) domain, // Domain number, AArch32 only
        bits(2) errortype, // [Armv8.2 RAS] AArch32 AET or AArch64 SET
        bits(4) debugmoe) // Debug method of entry, from AArch32 only
```

## Library pseudocode for shared/functions/memory/EffectiveTCMAFullAddress

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bittype EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assertFullAddress is ( HaveELPASpace(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;paspacer,
    bits(52) address
)
```

## Library pseudocode for shared/functions/memory/**FaultHint\_Prefetch**

```

enumeration // Signals the memory system that memory accesses of type HINT to or from the specified address
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches. Fault {Hint_Prefetch(bits(64) address,
    Fault_AccessFlag,
    Fault_Alignment,
    Fault_Background,
    Fault_Domain,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_HWUpdateAccessFlag,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint}; hint, integer target, boolean stream);

```

## Library pseudocode for shared/functions/memory/**FaultRecordMBReqDomain**

```

type enumeration FaultRecord is (MBReqDomain {Fault statuscode, // Fault Status MBReqDomain_Nonshareable,
    AccType acctype, // Type of access that faulted MBReqDomain_InnerShareable,
    MBReqDomain_OuterShareable, FullAddress ipaddress, // Intermediate physical address
    boolean s2fslwalk, // Is on a Stage 1 translation table walk
    boolean write, // TRUE for a write, FALSE for a read
    integer level, // For translation, access flag and Permission faults
    bit extflag, // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean secondstage, // Is a Stage 2 abort
    bits(4) domain, // Domain number, AArch32 only
    bits(2) errortype, // [Armv8.2 RAS] AArch32 AET or AArch64 SET
    bits(4) debugmoe) // Debug method of entry, from AArch32 only MBReqDomain_FullSy

```

## Library pseudocode for shared/functions/memory/**FullAddressMBReqTypes**

```

type enumeration FullAddress is (MBReqTypes {
    MBReqTypes_Reads, MBReqTypes_Writes, PAspace paspace,
    bits(52) address
) MBReqTypes_All};

```

## Library pseudocode for shared/functions/memory/**Hint\_PrefetchMPAM**

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.type
Hint_Prefetch(bits(64) address, PARTIDtype = bits(16);
type PMGtype = bits(8);
type PARTIDspaceType = bit;
constant PARTIDspaceType PidSpace_Secure = '0';
constant PARTIDspaceType PidSpace_NonSecure = '1';

type MPAMInfo is (
    PARTIDspaceType mpam_ns,
    PARTIDtype partid,
    PMGtypePrefetchHint hint, integer target, boolean stream);pmg
)
```

## Library pseudocode for shared/functions/memory/**IsDataAccessMemAttrHints**

```
// IsDataAccess()
// =====
// Return TRUE if access is to data memory.

boolean type IsDataAccess(MemAttrHints is (
    bits(2) attrs, // See MemAttr_*, Cacheability attributes
    bits(2) hints, // See MemHint_*, Allocation hints
    boolean transient
)AccType acctype)
    return !(acctype IN {AccType_IFETCH, AccType_TTW,
        AccType_DC, AccType_IC,
        AccType_AT, AccType_ATPAN
    });
```

## Library pseudocode for shared/functions/memory/**MBReqDomainMemType**

```
enumeration MBReqDomain {MemType {MBReqDomain_Nonshareable, MemType_Normal, MBReqDomain_InnerShareable,
    MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

## Library pseudocode for shared/functions/memory/**MBReqTypesMemoryAttributes**

```
enumeration type MBReqTypes {MemoryAttributes is (MBReqTypes_Reads, memtype, MBReqTypes_Writes, device,
    MemAttrHints outer, // Outer hints and attributes
    ShareabilityMBReqTypes_All};shareability, // Shareability attribute
boolean tagged, // Tagged access
bit xs // XS attribute
)
```

## Library pseudocode for shared/functions/memory/MPAMPASpace

```
type enumeration PARTIDtype = bits(16);
type PAspace { PMGtype = bits(8);

enumeration PAS_NonSecure, PARTIDspaceType {PAS_Secure,
};
    PIdSpace_Secure,
    PIdSpace_NonSecure
};

type MPAMInfo is (
    PARTIDspaceType mpam_sp,
    PARTIDtype partid,
    PMGtype pmg
)
```

## Library pseudocode for shared/functions/memory/MemAttrHintsPermissions

```
type MemAttrHints is (
    bits(2) attrs, // See MemAttr_*, Cacheability attributes
    bits(2) hints, // See MemHint_*, Allocation hints
    boolean transient
Permissions is (
    bits(2) ap_table, // Stage 1 hierarchical access permissions
    bit xn_table, // Stage 1 hierarchical execute-never for single EL regimes
    bit pxn_table, // Stage 1 hierarchical privileged execute-never
    bit ux_n_table, // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap, // Stage 1 access permissions
    bit xn, // Stage 1 execute-never for single EL regimes
    bit ux_n, // Stage 1 unprivileged execute-never
    bit pxn, // Stage 1 privileged execute-never
    bits(2) s2ap, // Stage 2 access permissions
    bit s2xn_x, // Stage 2 extended execute-never
    bit s2xn // Stage 2 execute-never
)
```

## Library pseudocode for shared/functions/memory/MemTypePhysMemRead

```
enumeration // Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an external abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.
(PhysMemRetStatus, bits(8*size)) MemType {PhysMemRead(MemType_Normal, desc, integer size, MemType_Device),
```

## Library pseudocode for shared/functions/memory/MemoryAttributesPhysMemRetStatus

```
type MemoryAttributes is (PhysMemRetStatus is (
    MemTypeFault memtype, statuscode, // Fault Status
    bit extflag, // IMPLEMENTATION DEFINED
    // syndrome for External aborts
    bits(2) errortype, // optional error state
    // returned on a physical
    // memory access
    bits(64) store64bstatus, // status of 64B store
    DeviceTypeAccType device, // For Device memory types
    MemAttrHints inner, // Inner hints and attributes
    MemAttrHints outer, // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    boolean tagged, // Tagged access
    bit xs // XS attribute
) acctype) // Type of access that faulted
```

### Library pseudocode for shared/functions/memory/**PASpacePhysMemWrite**

```
enumeration// Writes the value to memory, and returns the status of the write.
// If there is an external abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
PhysMemRetStatus PASpace {PhysMemWrite(
    PAS_NonSecure, desc, integer size,
    PAS_Secure,
}; accdesc,
    bits(8*size) value);
```

### Library pseudocode for shared/functions/memory/**PermissionsPrefetchHint**

```
typeenumeration Permissions is (
    bits(2) ap_table, // Stage 1 hierarchical access permissions
    bit xn_table, // Stage 1 hierarchical execute-never for single EL regimes
    bit pxn_table, // Stage 1 hierarchical privileged execute-never
    bit uxn_table, // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap, // Stage 1 access permissions
    bit xn, // Stage 1 execute-never for single EL regimes
    bit uxn, // Stage 1 unprivileged execute-never
    bit pxn, // Stage 1 privileged execute-never
    bits(2) s2ap, // Stage 2 access permissions
    bit s2xn, // Stage 2 extended execute-never
    bit s2xn // Stage 2 execute-never
)PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

### Library pseudocode for shared/functions/memory/**PhysMemReadShareability**

```
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.
(PhysMemRetStatus, bits(8*size))enumeration PhysMemRead(Shareability {AddressDescriptor desc, integer size,
    Shareability_ISH,
    AccessDescriptor accdesc); Shareability_OSH
};
```

### Library pseudocode for shared/functions/ memory/**PhysMemRetStatusSpeculativeStoreBypassBarrierToPA**

```
type PhysMemRetStatus is (Fault statuscode, // Fault Status
    bit extflag, // IMPLEMENTATION DEFINED
    bits(2) errortype, // optional error state
    bits(64) store64bstatus, // status of 64B store
    AccType acctype) // Type of access that faultedSpeculativeStoreBypassBarrierToPA
```

### Library pseudocode for shared/functions/ memory/**PhysMemWriteSpeculativeStoreBypassBarrierToVA**

```
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
    bits(8*size) value); SpeculativeStoreBypassBarrierToVA();
```

### Library pseudocode for shared/functions/memory/**PrefetchHintTag**

```
enumerationconstant integer PrefetchHint {LOG2_TAG_GRANULE = 4;
    constant integerPrefetch_READ, TAG_GRANULE = 1 << Prefetch_WRITE, Prefetch_EXEC};
```

## Library pseudocode for shared/functions/memorympam/ShareabilityDefaultMPAMInfo

```
enumeration// DefaultMPAMInfo()
// =====
// Returns default MPAM info. The partidspace argument sets
// the PARTID space of the default MPAM information returned.
MPAMInfo Shareability {DefaultMPAMInfo(
    Shareability_NSH,partidspace)
    Shareability_ISH,DefaultInfo;
    DefaultInfo.mpam_ns = partidspace;
    DefaultInfo.partid =
        .
        .
    DefaultInfo.pmg = DefaultPMGShareability_OSH
};
return DefaultInfo;
```

## Library pseudocode for shared/functions/memorympam/SpeculativeStoreBypassBarrierToPADefaultPARTID

```
constant PARTIDtype SpeculativeStoreBypassBarrierToPA();DefaultPARTID = 0<15:0>;
```

## Library pseudocode for shared/functions/memorympam/SpeculativeStoreBypassBarrierToVADefaultPMG

```
constant PMGtype SpeculativeStoreBypassBarrierToVA();DefaultPMG = 0<7:0>;
```

## Library pseudocode for shared/functions/memorympam/TagGenMPAMcurEL

```

constant integer // GenMPAMcurEL()
// =====
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo LOG2_TAG_GRANULE = 4;

constant integer GenMPAMcurEL( TAG_GRANULE = 1 <<acctype)
    bits(2) mpamEL;
    boolean validEL = FALSE; security = if IsSecure() then SS_Secure else SS_NonSecure;
    boolean InD = FALSE;
    PARTIDspaceType pspace = PARTIDspaceFromSS(security);
    if pspace == PIdSpace_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
        validEL = TRUE;
        case acctype of
            when AccType_IFETCH, AccType_IC
                InD = TRUE;
            otherwise
                // Other access types are DATA accesses
                InD = FALSE;
        if !validEL then
            return DefaultMPAMinfo(pspace);
        if HaveEMPAMExt() && security == SS_Secure then
            if MPAM3_EL3.FORCE_NS == '1' then
                pspace = PIdSpace_NonSecure;
            if MPAM3_EL3.SDEFLT == '1' then
                return DefaultMPAMinfo(pspace);
        if !MPAMisEnabled() then
            return DefaultMPAMinfo(pspace);
        else
            return genMPAMLOG2_TAG_GRANULESecurityState;(mpamEL, InD, pspace);

```

## Library pseudocode for shared/functions/mpam/DefaultMPAMInfoMAP\_vPARTID

```
// DefaultMPAMInfo()
// =====
// Returns default MPAM info. The partidspace argument sets
// the PARTID space of the default MPAM information returned.
// MAP_vPARTID()
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

MPAMInfo(PARTIDtype, boolean) DefaultMPAMInfo(MAP_vPARTID(PARTIDspaceTypePARTIDtype partidspace) vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    MPAMInfoPARTIDtype DefaultInfo;
    DefaultInfo.mpam_sp = partidspace;
    DefaultInfo.partid = ret;
    boolean err;
    integer virt = UInt(vpartid);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        DefaultInfo.pmg = err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = (MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTIDDefaultPMGUInt;
        return DefaultInfo; err = TRUE;
    return (ret, err);
```

## Library pseudocode for shared/functions/mpam/DefaultPARTIDMPAMisEnabled

```
constant PARTIDtype// MPAMisEnabled()
// =====
// Returns TRUE if MPAMisEnabled.

boolean DefaultPARTID = 0<15:0>;MPAMisEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

## Library pseudocode for shared/functions/mpam/DefaultPMGMPAMisVirtual

```
constant PMGtype// MPAMisVirtual()
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean DefaultPMG = 0<7:0>;MPAMisVirtual(bits(2) el)
return (MPAMIDR_EL1.HAS_HCR == '1' &&EL2Enabled() &&
((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
(HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
(el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

## Library pseudocode for shared/functions/mpam/GenMPAMcurELPARTIDspaceFromSS

```
// GenMPAMcurEL()
// =====
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.
// PARTIDspaceFromSS()
// =====
// Returns the primary PARTID space from the Security State.

MPAMinfoPARTIDspaceType GenMPAMcurEL(PARTIDspaceFromSS(AccType acctype)
bits(2) mpamEL;
boolean validEL = FALSE;
SecurityState security =security)
case security of
when CurrentSecurityStateSS_NonSecure();
boolean InD = FALSE;return
PARTIDspaceType pspace = PARTIDspaceFromSS(security);
if pspace == PIdSpace_NonSecure && !;
whenMPAMisEnabled() then
return DefaultMPAMinfo(pspace);
if UsingAArch32() then
(validEL, mpamEL) = ELFromM32(PSTATE.M);
else
mpamEL = if acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
validEL = TRUE;
case acctype of
when AccType_IFETCH, AccType_IC
InD = TRUE;
otherwise
// Other access types are DATA accesses
InD = FALSE;
if !validEL then
return DefaultMPAMinfo(pspace);
if HaveEMPAMExt() && security == SS_Secure then
if MPAM3_EL3.FORCE_NS == '1' then
pspace =return PIdSpace_NonSecurePIdSpace_Secure;
if MPAM3_EL3.SDEFLT == '1' then
returnotherwise DefaultMPAMinfoUnreachable(pspace);
if !MPAMisEnabled() then
return DefaultMPAMinfo(pspace);
else
return genMPAM(mpamEL, InD, pspace);();
```

## Library pseudocode for shared/functions/mpam/MAP\_vPARTIDgenMPAM

```
// MAP_vPARTID()
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.
// genMPAM()
// =====
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a PARTID in PARTID space pspace.

(PARTIDtype, boolean)MPAMinfo MAP_vPARTID(genMPAM(bits(2) el, boolean InD, PARTIDtypePARTIDspaceType vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.pspace)
    MPAMinfo returninfo;
    PARTIDtype ret;
    boolean err;
    integer virt =partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use EL1 the virtual machine's PARTIDs.
    boolean gstplk = (el == UIntEL0(vpartid));
    integer vpmrmax = && UIntEL2Enabled(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if() &&
        MPAMHCR_EL2.GSTAPP_PLK == '1' &&
        HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then UIntEL1(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret =else el;
    (partidel, perr) = mapvpmwgenPARTID(virt);
    err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret =(eff_el, InD); DefaultPARTIDPMGtype;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max =groupel = UIntgenPMG(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);(eff_el, InD, perr);
    returninfo.mpam_ns = pspace;
    returninfo.partid = partidel;
    returninfo.pmg = groupel;
    return returninfo;
```

## Library pseudocode for shared/functions/mpam/MPAMisEnabledgenMPAMel

```
// MPAMisEnabled()
// =====
// Returns TRUE if MPAMisEnabled.
// genMPAMel()
// =====
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

boolean MPAMinfo MPAMisEnabled()
    el = genMPAMel(bits(2) el, boolean InD) HighestELSecurityState();
    case el of
        when security = EL3SecurityStateAtEL return MPAM3_EL3.MPAMEN == '1';
        when (el); EL2PARTIDspaceType return MPAM2_EL2.MPAMEN == '1';
        when space = (security);
    boolean use_default = !(HaveMPAMExt() && MPAMisEnabled());
    if HaveEMPAMExt() && security == SS_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            space = PIdSpace_NonSecure;
        if MPAM3_EL3.SDEFLT == '1' then
            use_default = TRUE;
    if !use_default then
        return genMPAM(el, InD, space);
    else
        return DefaultMPAMinfoEL1PARTIDspaceFromSS return MPAM1_EL1.MPAMEN == '1';(space);
```

## Library pseudocode for shared/functions/mpam/MPAMisVirtualgenPARTID

```
// MPAMisVirtual()
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.
// genPARTID()
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

boolean(PARTIDtype, boolean) MPAMisVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && genPARTID(bits(2) el, boolean InD) EL2EnabledPARTIDtype() &&
        ((el == partidel = EL0getMPAM_PARTID && MPAMHCR_EL2.EL0_VPMEN == '1' &&
            (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
            (el == (el, InD); partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTIDEL1PARTIDtype && MPAMHCR_EL2.EL1_VPMEN == '1')));(partidel);
    else
        return (partidel, FALSE);
```

## Library pseudocode for shared/functions/mpam/PARTIDspaceFromSSgenPMG

```
// PARTIDspaceFromSS()
// =====
// Returns the primary PARTID space from the Security State.
// genPMG()
// =====
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PARTIDspaceTypePMGtype PARTIDspaceFromSS(genPMG(bits(2) el, boolean InD, boolean partid_err)
integer pmg_max = SecurityStateUInt security)
case security of
    when (MPAMIDR_EL1.PMG_MAX);
        // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
        // use the default or if it uses the PMG from getMPAM_PMG.
        if partid_err then
            return SS_NonSecureDefaultPMG
        return; PIdSpace_NonSecurePMGtype;
    when groupel = SS_SecuregetMPAM_PMG
        return (el, InD);
    if PIdSpace_SecureUInt;
        otherwise (groupel) <= pmg_max then
            return groupel;
    return
    UnreachableDefaultPMG();;
```

## Library pseudocode for shared/functions/mpam/genMPAMgetMPAM\_PARTID

```
// genMPAM()
// =====
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a PARTID in PARTID space pspace.
// getMPAM_PARTID()
// =====
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

MPAMinfoPARTIDtype genMPAM(bits(2) el, boolean InD, getMPAM_PARTID(bits(2) MPAMn, boolean InD) PARTIDspace
MPAMinfo returninfo;
PARTIDtype partidel;
boolean perr;
// gstplk is guest OS application locked by the EL2 hypervisor to
// only use EL1 the virtual machine's PARTIDs.
boolean gstplk = (el == partid;
boolean el2avail = EL0 && EL2Enabled() &&
MPAMHCR_EL2.GSTAPP_PLK == '1' &&
HCR_EL2.TGE == '0');
bits(2) eff_el = if gstplk then();

if InD then
  case MPAMn of
    when '11' partid = MPAM3_EL3.PARTID_I;
    when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else EL1Zeros else el;
  (partidel, perr) =();
  when '01' partid = MPAM1_EL1.PARTID_I;
  when '00' partid = MPAM0_EL1.PARTID_I;
  otherwise partid = genPARTIDPARTIDtype(eff_el, InD);UNKNOWN;
else
  case MPAMn of
    when '11' partid = MPAM3_EL3.PARTID_D;
    when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else
PMGtypeZeros groupel =();
    when '01' partid = MPAM1_EL1.PARTID_D;
    when '00' partid = MPAM0_EL1.PARTID_D;
    otherwise partid = genPMGPARTIDtype(eff_el, InD, perr);
  returninfo.mpam_sp = pspace;
  returninfo.partid = partidel;
  returninfo.pmg = groupel;
  return returninfo;UNKNOWN;
  return partid;
```

## Library pseudocode for shared/functions/mpam/genMPAMelgetMPAM\_PMG

```
// genMPAMel()
// =====
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.
// getMPAM_PMG()
// =====
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

MPAMinfoPMGtype genMPAMel(bits(2) el, boolean InD) getMPAM_PMG(bits(2) MPAMn, boolean InD)
    SecurityStatePMGtype security = pmg;
    boolean el2avail = SecurityStateAtEL2Enabled(el);

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else
                PARTIDspaceTypeZeros space = ();
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PARTIDspaceFromSSPMGtype(security);
        boolean use_default = !(UNKNOWN;
    else
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_D;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else HaveMPAMExtZeros() &&();
            when '01' pmg = MPAM1_EL1.PMG_D;
            when '00' pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = MPAMisEnabledPMGtype();
        if HaveEMPAMExt() && security == SS_Secure then
            if MPAM3_EL3.FORCE_NS == '1' then
                space = PIdSpace_NonSecure;
            if MPAM3_EL3.SDEFLT == '1' then
                use_default = TRUE;
        if !use_default then
            return genMPAM(el, InD, space);
        else
            return DefaultMPAMinfo(space);UNKNOWN;
    return pmg;
```

## Library pseudocode for shared/functions/mpam/genPARTIDmapvpmw

```
// genPARTID()
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.
// mapvpmw()
// =====
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

(PARTIDtype, boolean)PARTIDtype genPARTID(bits(2) el, boolean InD)mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw =
PARTIDtypeZeros partidel = getMPAM_PARTID(el, InD);
PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
if UInt(partidel) > UInt(partid_max) then
    return (DefaultPARTID, TRUE);
if MPAMisVirtual(el) then
    return MAP_vPARTID(partidel);
else
    return (partidel, FALSE);(64);
// vpme_lsb selects LSB of field within register
integer vpme_lsb = (vpartid MOD 4) * 16;
return vpmw<vpme_lsb +: 16>;
```

```

// genPMG()
// =====
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.
// ASID[]
// =====
// Effective ASID.

PMGtypebits(16) genPMG(bits(2) el, boolean InD, boolean partid_err)
integer pmg_max =ASID[]
if UIntEL2Enabled(MPAMIDR_EL1.PMG_MAX);
// It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
// use the default or if it uses the PMG from getMPAM_PMG.
if partid_err then
    return() && ! DefaultPMGELUsingAArch32;(
    PMGtypeEL2 groupel =) && HCR_EL2.<E2H, TGE> == '11' then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;

    elseif ! getMPAM_PMGELUsingAArch32(el, InD);
    if( UIntEL1(groupel) <= pmg_max then
        return groupel;
    return) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;

    else
        if TTBCR.EAE == '0' then
            return (CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtendDefaultPMGZeroExtend;(TTBR0.ASID, 16);

```

**Library pseudocode for shared/  
functions/~~mpampredictionrestrict/~~getMPAM\_PARTIDExecutionCntxt**

```
// getMPAM_PARTID()
// =====
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD)ExecutionCntxt is (
    boolean      is_vmid_valid, // is vmid valid for current context
    boolean      all_vmid,      // should the operation be applied for all vmids
    bits(16)     vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean      is_asid_valid, // is asid valid for current context
    boolean      all_asid,      // should the operation be applied for all asids
    bits(16)     asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)      target_el,     // target EL at which operation is performed
    PARTIDtypeSecurityState partid;
    boolean el2avail =security, EL2EnabledRestrictType();

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
        else
            case MPAMn of
                when '11' partid = MPAM3_EL3.PARTID_D;
                when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
                when '01' partid = MPAM1_EL1.PARTID_D;
                when '00' partid = MPAM0_EL1.PARTID_D;
                otherwise partid = PARTIDtype UNKNOWN;
            return partid;restriction // type of restriction operation
        )
    )
```

## Library pseudocode for shared/ functions/~~mpampredictionrestrict~~/getMPAM\_PMGRESTRICT\_PREDICTIONS

```
// getMPAM_PMG()
// =====
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype// RESTRICT_PREDICTIONS()
// =====
// Clear all speculated values. getMPAM_PMG(bits(2) MPAMn, boolean InD)RESTRICT_PREDICTIONS(
    PMGtypeExecutionCntxt pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
        else
            case MPAMn of
                when '11' pmg = MPAM3_EL3.PMG_D;
                when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
                when '01' pmg = MPAM1_EL1.PMG_D;
                when '00' pmg = MPAM0_EL1.PMG_D;
                otherwise pmg = PMGtype UNKNOWN;
            return pmg;e);
IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/~~mpampredictionrestrict~~/mapvpmwRestrictType

```
// mapvpmw()
// =====
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtypeenumeration mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw =RestrictType { RestrictType_DataValue,
RestrictType_ControlFlow,
Zeros(64);
// vpme_lsb selects LSB of field within register
integer vpme_lsb = (vpartid MOD 4) * 16;
return vpmw<vpme_lsb +: 16>;RestrictType_CachePrefetch
};
```

## Library pseudocode for shared/functions/predictionrestrict/ASIDTargetSecurityState

```
// ASID[]
// =====
// Effective ASID.
// TargetSecurityState()
// =====
// Decode the target security state for the prediction context.

bits(16)SecurityState ASID[]
    ifTargetSecurityState(bit NS)
    curr_ss = EL2EnabledSecurityStateAtEL() && !(PSTATE.EL);
    if curr_ss == ELUsingAArch32SS_NonSecure then
        return EL2SS_NonSecure() && HCR_EL2.<E2H, TGE> == '11' then
            if TCR_EL2.A1 == '1' then
                return TTBR1_EL2.ASID;
            else
                return TTBR0_EL2.ASID;

    elseif !;
    elseif curr_ss == ELUsingAArch32SS_Secure then
        case NS of
            when '0' return EL1SS_Secure() then
                if TCR_EL1.A1 == '1' then
                    return TTBR1_EL1.ASID;
                else
                    return TTBR0_EL1.ASID;

    else
        if TTBCR.EAE == '0' then
            return;
        when '1' return ZeroExtendSS_NonSecure(CONTEXTIDR.ASID, 16);
    else
        if TTBCR.A1 == '1' then
            return ZeroExtend(TTBR1.ASID, 16);
        else
            return ZeroExtend(TTBR0.ASID, 16);;
```

## Library pseudocode for shared/functions/predictionrestrictregisters/ExecutionCntxtBranchTo

```
type// BranchTo()
// =====
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits. ExecutionCntxt is (
    boolean is_vmid_valid, // is vmid valid for current context
    boolean all_vmid, // should the operation be applied for all vmids
    bits(16) vmid, // if all_vmid = FALSE, vmid to which operation is applied
    boolean is_asid_valid, // is asid valid for current context
    boolean all_asid, // should the operation be applied for all asids
    bits(16) asid, // if all_asid = FALSE, ASID to which operation is applied
    bits(2) target_el, // target EL at which operation is performed
    SecurityStateBranchType security, branch_type, boolean branch_conditional)
(branch_type);
if N == 32 then
    assert UsingAArch32();
    _PC = ZeroExtend(target);
else
    assert N == 64 && !UsingAArch32();
    bits(64) target_vaddress = AArch64.BranchAddrRestrictTypeHint_Branch restriction // type c
)(target<63:0>);
    _PC = target_vaddress;
return;
```

## Library pseudocode for shared/ functions/predictionrestrictregisters/RESTRICT\_PREDICTIONSBranchToAddr

```
// RESTRICT_PREDICTIONS()
// =====
// Clear all speculated values.// BranchToAddr()
// =====
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

RESTRICT_PREDICTIONS(BranchToAddr(bits(N) target, branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32ExecutionCntxtBranchType c)
    IMPLEMENTATION_DEFINED;();
    PC = target<63:0>;
    return;
```

## Library pseudocode for shared/functions/predictionrestrictregisters/RestrictTypeBranchType

```
enumeration RestrictType {BranchType {
    RestrictType_DataValue,BranchType_DIRCALL, // Direct Branch with link
    RestrictType_ControlFlow,BranchType_INDCALL, // Indirect Branch with link
    RestrictType_CachePrefetch
};BranchType_ERET, // Exception return (indirect)BranchType_DBGEXIT, // Exit from Debug state
BranchType_RET, // Indirect branch with function return hint
BranchType_DIR, // Direct branch
BranchType_INDIR, // Indirect branch
BranchType_EXCEPTION, // Exception entry
BranchType_RESET, // Reset
BranchType_UNKNOWN}; // Other
```

## Library pseudocode for shared/ functions/predictionrestrictregisters/TargetSecurityStateHint\_Branch

```
// TargetSecurityState()
// =====
// Decode the target security state for the prediction context.

SecurityState// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction. TargetSecurityState(bit NS)
    curr_ss =Hint_Branch( SecurityStateAtELBranchType(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;hint);
```

### Library pseudocode for shared/functions/registers/**BranchToNextInstrAddr**

```
// BranchTo()  
// =====  
// Set program counter to a new address, with a branch type.  
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.  
// In AArch64 state the address might include a tag in the top eight bits. // Return address of the sequen  
bits(N)  
  
BranchTo(bits(N) target, NextInstrAddr(); BranchType branch_type, boolean branch_conditional)  
    Hint_Branch(branch_type);  
    if N == 32 then  
        assert UsingAArch32();  
        _PC = ZeroExtend(target);  
    else  
        assert N == 64 && !UsingAArch32();  
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>);  
  
        _PC = target_vaddress;  
    return;
```

### Library pseudocode for shared/functions/registers/**BranchToAddrResetExternalDebugRegisters**

```
// BranchToAddr()  
// =====  
// Set program counter to a new address, with a branch type.  
// In AArch64 state the address does not include a tag in the top eight bits. // Reset the External Debug  
  
BranchToAddr(bits(N) target, ResetExternalDebugRegisters(boolean cold_reset); BranchType branch_type)  
    Hint_Branch(branch_type);  
    if N == 32 then  
        assert UsingAArch32();  
        _PC = ZeroExtend(target);  
    else  
        assert N == 64 && !UsingAArch32();  
        _PC = target<63:0>;  
    return;
```

### Library pseudocode for shared/functions/registers/**BranchTypeThisInstrAddr**

```
enumeration // ThisInstrAddr()  
// =====  
// Return address of the current instruction.  
  
bits(N) BranchType {ThisInstrAddr()  
    assert N == 64 || (N == 32 &&  
        BranchType_DIRCALL, // Direct Branch with link  
        BranchType_INDCALL, // Indirect Branch with link  
        BranchType_ERET, // Exception return (indirect)  
        BranchType_DBGEXIT, // Exit from Debug state  
        BranchType_RET, // Indirect branch with function return hint  
        BranchType_DIR, // Direct branch  
        BranchType_INDIR, // Indirect branch  
        BranchType_EXCEPTION, // Exception entry  
        BranchType_RESET, // Reset  
        BranchType_UNKNOWN}; // Other()  
    return _PC<N-1:0>;
```

### Library pseudocode for shared/functions/registers/**Hint\_Branch\_PC**

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing  
// the next instruction. bits(64) _PC;  
Hint_Branch(BranchType hint);
```

## Library pseudocode for shared/functions/registers/NextInstrAddr\_R

```
// Return address of the sequentially next instruction.  
bits(N) array bits(64) _R[0..30]; NextInstrAddr();
```

## Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters\_V

```
// Reset the External Debug registers in the Core power domain. array bits(128) _V[0..31];  
ResetExternalDebugRegisters(boolean cold_reset);
```

## Library pseudocode for shared/functions/registers/sysregisters/ThisInstrAddrSPSR

```
// ThisInstrAddr()  
// =====  
// Return address of the current instruction.  
// SPSR[] - non-assignment form  
// =====  
  
bits(N) ThisInstrAddr()  
    assert N == 64 || (N == 32 && SPSR[]  
    bits(N) result;  
    if UsingAArch32();  
    return PC<N-1:0>;() then  
        assert N == 32;  
        case PSTATE.M of  
            when M32_FIQ      result = SPSR_fiq<N-1:0>;  
            when M32_IRQ      result = SPSR_irq<N-1:0>;  
            when M32_Svc      result = SPSR_svc<N-1:0>;  
            when M32_Monitor  result = SPSR_mon<N-1:0>;  
            when M32_Abort    result = SPSR_abt<N-1:0>;  
            when M32_Hyp      result = SPSR_hyp<N-1:0>;  
            when M32_Undef    result = SPSR_und<N-1:0>;  
            otherwise         Unreachable();  
    else  
        assert N == 64;  
        case PSTATE.EL of  
            when EL1          result = SPSR_EL1<N-1:0>;  
            when EL2          result = SPSR_EL2<N-1:0>;  
            when EL3          result = SPSR_EL3<N-1:0>;  
            otherwise         Unreachable();  
    return result;  
  
// SPSR[] - assignment form  
// =====  
  
SPSR[] = bits(N) value  
    if UsingAArch32() then  
        assert N == 32;  
        case PSTATE.M of  
            when M32_FIQ      SPSR_fiq = ZeroExtend(value);  
            when M32_IRQ      SPSR_irq = ZeroExtend(value);  
            when M32_Svc      SPSR_svc = ZeroExtend(value);  
            when M32_Monitor  SPSR_mon = ZeroExtend(value);  
            when M32_Abort    SPSR_abt = ZeroExtend(value);  
            when M32_Hyp      SPSR_hyp = ZeroExtend(value);  
            when M32_Undef    SPSR_und = ZeroExtend(value);  
            otherwise         Unreachable();  
    else  
        assert N == 64;  
        case PSTATE.EL of  
            when EL1          SPSR_EL1 = ZeroExtend(value);  
            when EL2          SPSR_EL2 = ZeroExtend(value);  
            when EL3          SPSR_EL3 = ZeroExtend(value);  
            otherwise         Unreachable();  
    return;
```

## Library pseudocode for shared/functions/**registerssystem/PCArchVersion**

```
bits(64) _PC; enumeration ArchVersion {  
    ARMv8p0  
    , ARMv8p1  
    , ARMv8p2  
    , ARMv8p3  
    , ARMv8p4  
    , ARMv8p5  
    , ARMv8p6  
    , ARMv8p7  
    , ARMv8p8  
};
```

## Library pseudocode for shared/functions/**registerssystem/RBranchTargetCheck**

```
array bits(64) _R[0..30]; // BranchTargetCheck()  
// =====  
// This function is executed checks if the current instruction is a valid target for a branch  
// taken into, or inside, a guarded page. It is executed on every cycle once the current  
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been  
// determined for the current instruction. BranchTargetCheck()  
assert HaveBTIExt() && !UsingAArch32();  
  
// The branch target check considers two state variables:  
// * InGuardedPage, which is evaluated during instruction fetch.  
// * BTypeCompatible, which is evaluated during instruction decode.  
if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then  
    bits(64) pc = ThisInstrAddr();  
    AArch64.BranchTargetException(pc<51:0>);  
  
boolean branch_instr = AArch64.ExecutingBR0rBLR0rRetInstr();  
boolean bti_instr = AArch64.ExecutingBTIInstr();  
  
// PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.  
if !(branch_instr || bti_instr) then  
    BTypeNext = '00';
```

## Library pseudocode for shared/functions/**registerssystem/VClearEventRegister**

```
array bits(128) _V[0..31]; // ClearEventRegister()  
// =====  
// Clear the Event Register of this PE. ClearEventRegister()  
EventRegister = '0';  
return;
```

## Library pseudocode for shared/functions/sysregisters/system/SPSRClearPendingPhysicalSError

```
// SPSR[] - non-assignment form
// =====

bits(N) // Clear a pending physical SError interrupt. SPSR[]
    bits(N) result;
    if ClearPendingPhysicalSError(); UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ      result = SPSR_fiq<N-1:0>;
            when M32_IRQ      result = SPSR_irq<N-1:0>;
            when M32_Svc      result = SPSR_svc<N-1:0>;
            when M32_Monitor  result = SPSR_mon<N-1:0>;
            when M32_Abort    result = SPSR_abt<N-1:0>;
            when M32_Hyp      result = SPSR_hyp<N-1:0>;
            when M32_Undef    result = SPSR_und<N-1:0>;
            otherwise        Unreachable();
        else
            assert N == 64;
            case PSTATE.EL of
                when EL1      result = SPSR_EL1<N-1:0>;
                when EL2      result = SPSR_EL2<N-1:0>;
                when EL3      result = SPSR_EL3<N-1:0>;
                otherwise      Unreachable();
            return result;

// SPSR[] - assignment form
// =====

SPSR[] = bits(N) value
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ      SPSR_fiq = ZeroExtend(value);
            when M32_IRQ      SPSR_irq = ZeroExtend(value);
            when M32_Svc      SPSR_svc = ZeroExtend(value);
            when M32_Monitor  SPSR_mon = ZeroExtend(value);
            when M32_Abort    SPSR_abt = ZeroExtend(value);
            when M32_Hyp      SPSR_hyp = ZeroExtend(value);
            when M32_Undef    SPSR_und = ZeroExtend(value);
            otherwise        Unreachable();
        else
            assert N == 64;
            case PSTATE.EL of
                when EL1      SPSR_EL1 = ZeroExtend(value);
                when EL2      SPSR_EL2 = ZeroExtend(value);
                when EL3      SPSR_EL3 = ZeroExtend(value);
                otherwise      Unreachable();
            return;
```

## Library pseudocode for shared/functions/system/ArchVersionClearPendingVirtualSError

```
enumeration // Clear a pending virtual SError interrupt. ArchVersion {ClearPendingVirtualSError();
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
    , ARMv8p4
    , ARMv8p5
    , ARMv8p6
    , ARMv8p7
    , ARMv8p8
};
```

## Library pseudocode for shared/functions/system/**BranchTargetCheckConditionHolds**

```
// BranchTargetCheck()
// =====
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

boolean

BranchTargetCheck()
    assertConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1'); // EQ or NE
        when '001' result = (PSTATE.C == '1'); // CS or CC
        when '010' result = (PSTATE.N == '1'); // MI or PL
        when '011' result = (PSTATE.V == '1'); // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' result = TRUE; // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result; HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr();
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBR0rBLR0rRetInstr();
    boolean bti_instr = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

## Library pseudocode for shared/functions/system/**ClearEventRegisterConsumptionOfSpeculativeDataBarrier**

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;ConsumptionOfSpeculativeDataBarrier();
```

## Library pseudocode for shared/functions/system/ClearPendingPhysicalErrorCurrentInstrSet

```
// Clear a pending physical SError interrupt.// CurrentInstrSet()
// =====
InstrSet
ClearPendingPhysicalError();CurrentInstrSet()InstrSet result;
if UsingAArch32() then
    result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
    // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
else
    result = InstrSet_A64;
return result;
```

## Library pseudocode for shared/functions/system/ClearPendingVirtualErrorCurrentPL

```
// Clear a pending virtual SError interrupt.// CurrentPL()
// =====
PrivilegeLevel
ClearPendingVirtualError();CurrentPL()
return PLOfEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/ConditionHoldsCurrentSecurityState

```
// ConditionHolds()
// =====
// Return TRUE iff COND currently holds
// CurrentSecurityState()
// =====
// Returns the effective security state at the exception level based off current settings.

booleanSecurityState ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1'); // EQ or NE
        when '001' result = (PSTATE.C == '1'); // CS or CC
        when '010' result = (PSTATE.N == '1'); // MI or PL
        when '011' result = (PSTATE.V == '1'); // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' result = TRUE; // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrierDSBAlias

```
enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, ConsumptionOfSpeculativeDataBarrier();DSBAlias_DSB};
```

## Library pseudocode for shared/functions/system/CurrentInstrSetEL0

```
// CurrentInstrSet()
// =====

InstrSet constant bits(2) CurrentInstrSet() EL3 = '11';
constant bits(2)
    InstrSet result;
    if EL2 = '10';
constant bits(2) UsingAArch32() then
    result = if PSTATE.T == '0' then EL1 = '01';
constant bits(2) InstrSet_A32 else InstrSet_T32;
    // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
else
    result = InstrSet_A64;
return result; EL0 = '00';
```

## Library pseudocode for shared/functions/system/CurrentPLEL2Enabled

```
// CurrentPL()
// =====
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

PrivilegeLevel boolean CurrentPL()
EL2Enabled()
    return (EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecureEL2EnabledPL0fELHaveEL(PSTATE.EL));
```

## Library pseudocode for shared/functions/system/CurrentSecurityStateELFromM32

```
// CurrentSecurityState()
// =====
// Returns the effective security state at the exception level based off current settings.
// ELFromM32()
// =====

SecurityState(boolean, bits(2)) CurrentSecurityState()
    return ELFromM32(bits(5) mode);
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid, EL):
    // 'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    // and the current value of SCR.NS/SCR_EL3.NS.
    // 'EL' is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = ! (mode); // Check for modes that are not valid for this implementation
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0SecurityStateAtELBadMode(PSTATE.EL);
        otherwise
            valid = FALSE; // Passed an illegal mode value
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

## Library pseudocode for shared/functions/system/DSBAliasELFromSPSR

```

enumeration// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) DSBAlias {ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    if spsr<4> == '0' then // AArch64 state
        el = spsr<3:2>;
        if !DSBAlias SSBB,() then // No AArch64 support
            valid = FALSE;
        elsif !DSBAlias PSSBB,(el) then // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then // M[1] must be 0
            valid = FALSE;
        elsif el == && spsr<0> == '1' then // for EL0, M[0] must be 0
            valid = FALSE;
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
            valid = FALSE; // Unless Secure EL2 is enabled, EL2 only valid in Non-secure state
        else
            valid = TRUE;
    elsif HaveAArch32() then // AArch32 state
        (valid, el) = ELFromM32DSBAlias_DSB;({spsr<4:0>});
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
}

```

## Library pseudocode for shared/functions/system/EL0ELIsInHost

```

constant bits(2)// ELIsInHost()
// =====

boolean EL3 = '11';
constant bits(2)ELIsInHost(bits(2) el)
    if ! EL2 = '10';
constant bits(2){} || EL1 = '01';
constant bits(2){ } then
    return FALSE;
case el of
    when EL3
        return FALSE;
    when EL2
        return EL2Enabled() && HCR_EL2.E2H == '1';
    when EL1
        return FALSE;
    when EL0
        return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
    otherwise
        UnreachableEL0 = '00';{};

```

## Library pseudocode for shared/functions/system/EL2EnabledELStateUsingAArch32

```
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.
// ELStateUsingAArch32()
// =====

boolean EL2Enabled()
    return ELStateUsingAArch32(bits(2) el, boolean secure)
// See ELStateUsingAArch32K() for description. Must only be called in circumstances where
// result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
(known, aarch32) = HaveELStateUsingAArch32K(EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecure)
assert known;
return aarch32;
```

## Library pseudocode for shared/functions/system/EL3SDDTrapPriorityELStateUsingAArch32K

```
// EL3SDDTrapPriority()
// =====
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has the priority of the original trap exception.
// ELStateUsingAArch32K()
// =====

boolean(boolean,boolean) EL3SDDTrapPriority()
    return (ELStateUsingAArch32K(bits(2) el, boolean secure)
// Returns (known, aarch32):
// 'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
// using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
// 'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
if !HaltedHaveAArch32EL() && EDSCR.SDD == '1' &&
    boolean IMPLEMENTATION_DEFINED "(el) then
        return (TRUE, FALSE); // Exception level is using AArch64
elseif secure && el == EL2 then
    return (TRUE, FALSE); // Secure EL2 is using AArch64
elseif !HaveAArch64() then
    return (TRUE, TRUE); // Highest Exception level, and therefore all levels are using AArch32

// Remainder of function deals with the interprocessing cases when highest Exception level is using AArch64

boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;

aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '1')
aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && !HaveVirtHostExt() && !HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt()))
if el == EL0 && !aarch32_at_el1 then // Only know if EL0 using AArch32 from PSTATE
    if PSTATE.EL == EL0 then
        aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
    else
        known = FALSE; // EL0 state is UNKNOWN
else
    aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0} trap priority when EL3 is not implemented)

if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);
```

## Library pseudocode for shared/functions/system/ELFromM32ELUsingAArch32

```
// ELFromM32()
// =====
// ELUsingAArch32()
// =====

(boolean, bits(2)) boolean ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid, EL):
    // 'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    // and the current value of SCR.NS/SCR_EL3.NS.
    // 'EL' is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !ELUsingAArch32(bits(2) el);
    return BadModeELStateUsingAArch32(mode); // Check for modes that are not valid for this implementation
    case mode of
        when {el, M32_MonitorIsSecureBelowEL3}
            el = EL3;
        when M32_Hyp
            el = EL2;
        valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE; // Passed an illegal mode value
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);();
```

## Library pseudocode for shared/functions/system/~~ELFromSPSR~~~~ELUsingAArch32K~~

```
// ELFromSPSR()
// =====
// ELUsingAArch32K()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) (boolean, boolean) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    if spsr<4> == '0' then // AArch64 state
        el = spsr<3:2>;
        if !ELUsingAArch32K(bits(2) el)
            return HaveAArch64ELStateUsingAArch32K() then // No AArch64 support
                valid = FALSE;
            elsif !(el, HaveELIsSecureBelowEL3(el) then // Exception level not implemented
                valid = FALSE;
            elsif spsr<1> == '1' then // M[1] must be 0
                valid = FALSE;
            elsif el == EL0 && spsr<0> == '1' then // for EL0, M[0] must be 0
                valid = FALSE;
            elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
                valid = FALSE; // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
            else
                valid = TRUE;
            elsif HaveAArch32() then // AArch32 state
                (valid, el) = ELFromM32(spsr<4:0>);
            else
                valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);();
```

## Library pseudocode for shared/functions/system/~~ELIsInHost~~~~EffectiveTGE~~

```
// ELIsInHost()
// =====
// EffectiveTGE()
// =====
// Returns effective TGE value

boolean bit ELIsInHost(bits(2) el)
    if !EffectiveTGE()
        if HaveVirtHostExtEL2Enabled() || (;) then
            return if ELUsingAArch32(EL2) then
                return FALSE;
            case el of
                when EL3
                    return FALSE;
                when EL2
                    return EL2Enabled() && HCR_EL2.E2H == '1';
                when EL1
                    return FALSE;
                when EL0
                    return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
                otherwise
                    Unreachable();(); then HCR.TGE else HCR_EL2.TGE;
    else
        return '0'; // Effective value of TGE is zero
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32EndOfInstruction

```
// ELStateUsingAArch32()
// =====

boolean// Terminate processing of the current instruction. ELStateUsingAArch32(bits(2) el, boolean secure)
// See ELStateUsingAArch32K() for description. Must only be called in circumstances where
// result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
(known, aarch32) =EndOfInstruction(); ELStateUsingAArch32K(el, secure);
assert known;
return aarch32;
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32KEnterLowPowerState

```
// ELStateUsingAArch32K()
// =====

(boolean,boolean)// PE enters a low-power state. ELStateUsingAArch32K(bits(2) el, boolean secure)
// Returns (known, aarch32):
// 'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
// using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
// 'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
if !EnterLowPowerState();HaveAArch32EL(el) then
    return (TRUE, FALSE); // Exception level is using AArch64
elseif secure && el == EL2 then
    return (TRUE, FALSE); // Secure EL2 is using AArch64
elseif !HaveAArch64() then
    return (TRUE, TRUE); // Highest Exception level, and therefore all levels a

// Remainder of function deals with the interprocessing cases when highest Exception level is using A

boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;

aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EE
aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) &&
    ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && H
    !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt

if el == EL0 && !aarch32_at_el1 then // Only know if EL0 using AArch32 from PSTATE
    if PSTATE.EL == EL0 then
        aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
    else
        known = FALSE; // EL0 state is UNKNOWN
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});

if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);
```

## Library pseudocode for shared/functions/system/ELUsingAArch32EventRegister

```
// ELUsingAArch32()
// =====

booleanbits(1) EventRegister; ELUsingAArch32(bits(2) el)
return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

## Library pseudocode for shared/functions/system/ELUsingAArch32KExceptionalOccurrenceTargetState

```
// ELUsingAArch32K()
// =====

(boolean,boolean)enumeration ELUsingAArch32K(bits(2) el)
returnExceptionalOccurrenceTargetState { ELStateUsingAArch32K(el, AArch32_NonDebugState, AArch64_NonDe
IsSecureBelowEL3());DebugState
};
```

## Library pseudocode for shared/functions/system/EffectiveTGEFIQPending

```
// EffectiveTGE()
// =====
// Returns effective TGE value

bit// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.
(boolean, boolean) EffectiveTGE()
    ifFIQPending(); EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0'; // Effective value of TGE is zero
```

## Library pseudocode for shared/functions/system/EndOfInstructionGetAccumulatedFPEExceptions

```
// Terminate processing of the current instruction.// Returns FP exceptions accumulated by the PE.
bits(8)
EndOfInstruction();GetAccumulatedFPEExceptions();
```

## Library pseudocode for shared/functions/system/EnterLowPowerStateGetPSRFromPSTATE

```
// PE enters a low-power state.// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(N)
EnterLowPowerState();GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
    if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
        assert N == 32;
    else
        assert N == 64;
    bits(N) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20> = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27> = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
        if HaveDITExt() then
            if targetELState == AArch32_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else //AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
            spsr<19:16> = PSTATE.GE;
            spsr<15:10> = PSTATE.IT<7:2>;
            spsr<9> = PSTATE.E;
            spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state
            spsr<5> = PSTATE.T;
            assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
            spsr<4:0> = PSTATE.M;
        else // AArch64 state
            if HaveMTEExt() then spsr<25> = PSTATE.TCO;
            if HaveDITExt() then spsr<24> = PSTATE.DIT;
            if HaveUAOExt() then spsr<23> = PSTATE.UAO;
            spsr<21> = PSTATE.SS;
            if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
            if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
            if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
            spsr<9:6> = PSTATE.<D,A,I,F>;
            spsr<4> = PSTATE.nRW;
            spsr<3:2> = PSTATE.EL;
            spsr<0> = PSTATE.SP;
    return spsr;
```

### Library pseudocode for shared/functions/system/EventRegisterHasArchVersion

```
bits(1) EventRegister; // HasArchVersion()
// =====
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

### Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetStateHaveAArch32

```
enumeration // HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

boolean ExceptionalOccurrenceTargetState {HaveAArch32()
    return boolean IMPLEMENTATION_DEFINED "AArch32 state is supported at at least
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};"
```

### Library pseudocode for shared/functions/system/FIQPendingHaveAArch32EL

```
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.
(boolean, boolean) // HaveAArch32EL()
// =====

boolean FIQPending(); HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE; // The Exception level is not implemented
    elseif !HaveAArch32() then
        return FALSE; // No Exception level can use AArch32
    elseif !HaveAArch64() then
        return TRUE; // All Exception levels are using AArch32
    elseif el == HighestEL() then
        return FALSE; // The highest Exception level is using AArch64
    elseif el == EL0 then
        return TRUE; // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

### Library pseudocode for shared/functions/system/GetAccumulatedFPEExceptionsHaveAArch64

```
// Returns FP exceptions accumulated by the PE.
bits(8) // HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

boolean GetAccumulatedFPEExceptions(); HaveAArch64()
    return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64";
```

## Library pseudocode for shared/functions/system/GetPSRFromPSTATEHaveEL

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

bits(N) boolean GetPSRFromPSTATE(HaveEL(bits(2) el)
if el IN {ExceptionalOccurrenceTargetStateEL1 targetELState)
if, UsingAArch32EL0() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
    assert N == 32;
else
    assert N == 64;
bits(N) spsr = Zeros();
spsr<31:28> = PSTATE.<N,Z,C,V>;
if HavePANExt() then spsr<22> = PSTATE.PAN;
spsr<20> = PSTATE.IL;
if PSTATE.nRW == '1' then // AArch32 state
    spsr<27> = PSTATE.Q;
    spsr<26:25> = PSTATE.IT<1:0>;
    if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
    if HaveDITEExt() then
        if targetELState == AArch32_NonDebugState then
            spsr<21> = PSTATE.DIT;
        else // AArch64_NonDebugState or DebugState
            spsr<24> = PSTATE.DIT;
    if targetELState IN {AArch64_NonDebugState, DebugState} then
        spsr<21> = PSTATE.SS;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9> = PSTATE.E;
        spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state
        spsr<5> = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
        spsr<4:0> = PSTATE.M;
    else // AArch64 state
        if HaveMTEExt() then spsr<25> = PSTATE.TCO;
        if HaveDITEExt() then spsr<24> = PSTATE.DIT;
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        spsr<21> = PSTATE.SS;
        if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
        if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
        if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6> = PSTATE.<D,A,I,F>;
        spsr<4> = PSTATE.nRW;
        spsr<3:2> = PSTATE.EL;
        spsr<0> = PSTATE.SP;
    return spsr;} then
    return TRUE; // EL1 and EL0 must exist
return boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HasArchVersionHaveELUsingSecurityState

```
// HasArchVersion()
// =====
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HasArchVersion(HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when ArchVersionEL3 version)
            return version == assert secure;
        return (EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3ARMv8p0HaveEL || boolean IMPLEMENTATION_DEFINED;) ||
                (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

## Library pseudocode for shared/functions/system/HaveAArch32HaveFP16Ext

```
// HaveAArch32()
// HaveFP16Ext()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.
// Return TRUE if FP16 extension is supported

boolean HaveAArch32()
    return boolean IMPLEMENTATION_DEFINED "AArch32 state is supported at at leastHaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED; EL0";
```

## Library pseudocode for shared/functions/system/HaveAArch32ELHighestEL

```
// HaveAArch32EL()
// =====
// HighestEL()
// =====
// Returns the highest implemented Exception level.

booleanbits(2) HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HighestEL()
    if HaveEL(el) then
        return FALSE; // The Exception level is not implemented
    elsif !(HaveAArch32EL3() then
        return FALSE; // No Exception level can use AArch32
    elsif !) then
        return HaveAArch64EL3() then
        return TRUE; // All Exception levels are using AArch32
    elsif el ==;
    elsif HighestELHaveEL() then
        return FALSE; // The highest Exception level is using AArch64
    elsif el == ( ) then
        return EL2;
    else
        return EL1EL0EL2 then
        return TRUE; // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HaveAArch64Hint\_DGH

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

boolean// Provides a hint to close any gathering occurring within the micro-architecture. HaveAArch64()
return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64";Hint_DGH();
```

## Library pseudocode for shared/functions/system/HaveELHint\_WFE

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean// Hint_WFE()
// =====
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value. HaveEL(bits(2) el)
if el IN {Hint_WFE(integer localtimeout, WfxType wfxtype)
if IsEventRegisterSet() then
ClearEventRegister();
elseif HaveFeatWFXT() && LocalTimeoutEvent(localtimeout) then
// No further operation if the local timeout has expired.
EndOfInstruction();
else
bits(2) target_el;
trap = FALSE;
if PSTATE.EL == EL0 then
// Check for traps described by the OS which may be EL1 or EL2.
if HaveTWEExt() then
sctlr = SCTLR[];
trap = sctlr.nTWE == '0';
target_el = EL1;
elseAArch64.CheckForWFXTrap(EL1, wfxtype);
if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
// Check for traps described by the Hypervisor.
if HaveTWEExt() then
trap = HCR_EL2.TWE == '1';
target_el = EL2;
else
AArch64.CheckForWFXTrap(EL2, wfxtype);

if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
// Check for traps described by the Secure Monitor.
if HaveTWEExt() then
trap = SCR_EL3.TWE == '1';
target_el = EL3;
else
AArch64.CheckForWFXTrap(EL3, wfxtype);

if trap && PSTATE.EL != EL3 then
(delay_enabled, delay) = WFETrapDelay(target_el); // (If trap delay is enabled, Delay amo
if !WaitForEventUntilDelay(delay_enabled, delay) then
// Event did not arrive before delay expired
AArch64.WFXTrap(wfxtype, target_el); // Trap WFE
else
WaitForEvent} then
return TRUE; // EL1 and EL0 must exist
return boolean IMPLEMENTATION_DEFINED(localtimeout);
```

## Library pseudocode for shared/functions/system/HaveELUsingSecurityStateHint\_WFI

```
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean// Hint_WFI()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFI, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value. HaveELUsingSecurityState(bits(2) el, boolean secure)

case el of
  when Hint_WFI(integer localtimeout, EL3WfxType
    assert secure;
    return wfxtype)
  if HaveELInterruptPending() || (EL3HaveFeatWFI);
    when() && EL2LocalTimeoutEvent
      if secure then
        return(localtimeout)) then
          // No further operation if an interrupt is pending or the local timeout has expired. HaveELEndOf
        else
          if PSTATE.EL == EL2EL0) && then
            // Check for traps described by the OS. HaveSecureEL2ExtAArch64.CheckForWfxTrap();
          else
            return( HaveELEL1(, wfxtype);
          if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWfxTrap(EL2);
          otherwise
            return (, wfxtype);
          if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWfxTrap(EL3, wfxtype);
          WaitForInterrupt) ||
            (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation")); (localtimeou
```

## Library pseudocode for shared/functions/system/HaveFP16ExtHint\_Yield

```
// HaveFP16Ext()
// =====
// Return TRUE if FP16 extension is supported

boolean// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance. HaveFP16Ext()
  return boolean IMPLEMENTATION_DEFINED; Hint_Yield();
```

## Library pseudocode for shared/functions/system/HighestELIRQPending

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2)// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.
(boolean, boolean) HighestEL()
  if IRQPending(); HaveEL(EL3) then
    return EL3;
  elsif HaveEL(EL2) then
    return EL2;
  else
    return EL1;
```

## Library pseudocode for shared/functions/system/Hint\_DGHIllegalExceptionReturn

```
// Provides a hint to close any gathering occurring within the micro-architecture. // IllegalExceptionReturn
// =====
boolean
Hint_DGH();IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) =ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

## Library pseudocode for shared/functions/system/**Hint\_WFEInstrSet**

```
// Hint_WFE()
// =====
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.enumeration

Hint_WFE(integer localtimeout,InstrSet { WfxType wfxtype)
    ifInstrSet_A64, IsEventRegisterSet() thenInstrSet_A32,
        ClearEventRegister();
    elsif HaveFeatWfxT() && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction();
    else
        bits(2) target_el;
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            if HaveTWEExt() then
                sctlr = SCTLr[];
                trap = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWfxTrap(EL1, wfxtype);
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            if HaveTWEExt() then
                trap = HCR_EL2.TWE == '1';
                target_el = EL2;
            else
                AArch64.CheckForWfxTrap(EL2, wfxtype);

        if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            if HaveTWEExt() then
                trap = SCR_EL3.TWE == '1';
                target_el = EL3;
            else
                AArch64.CheckForWfxTrap(EL3, wfxtype);

        if trap && PSTATE.EL != EL3 then
            (delay_enabled, delay) = WFETrapDelay(target_el); // (If trap delay is enabled, Delay amou
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired
                AArch64.WfxTrap(wfxtype, target_el); // Trap WFE
        else
            WaitForEvent(localtimeout);InstrSet_T32};
```

## Library pseudocode for shared/functions/system/**Hint\_WFI****InstructionSynchronizationBarrier**

```
// Hint_WFI()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WfxType wfxtype)
    if InterruptPending() || (HaveFeatWfxT() && LocalTimeoutEvent(localtimeout)) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch64.CheckForWfxTrap(EL1, wfxtype);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWfxTrap(EL2, wfxtype);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWfxTrap(EL3, wfxtype);
        WaitForInterrupt(localtimeout); InstructionSynchronizationBarrier();
```

## Library pseudocode for shared/functions/system/**Hint\_Yield****InterruptPending**

```
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance. // InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean
Hint_Yield(); InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                         IsPhysicalErrorPending());

    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1');
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```

## Library pseudocode for shared/functions/system/**IRQPending****IsASEInstruction**

```
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.
(boolean, boolean) // Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.
boolean IRQPending(); IsASEInstruction();
```

## Library pseudocode for shared/functions/system/**IllegalExceptionReturnIsCMOWControlledInstruction**

```
// IllegalExceptionReturn()
// =====

// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CIGDVAC, or DC CIGVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.
boolean IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = IsCMOWControlledInstruction(); ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

## Library pseudocode for shared/functions/system/**InstrSetIsEventRegisterSet**

```
enumeration // IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean InstrSet {IsEventRegisterSet()
    return EventRegister == '1'; InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

## Library pseudocode for shared/functions/system/**InstructionSynchronizationBarrierIsHighestEL**

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL InstructionSynchronizationBarrier();() == el;
```

## Library pseudocode for shared/functions/system/InterruptPendingIsInHost

```
// InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.
// IsInHost()
// =====

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IsInHost();
    return IRQPendingELIsInHost();
    (fiq_pending, -) = FIQPending();
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                         IsPhysicalSErrorPending());

    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IM0 == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1');
        boolean vfiq_pending = HCR_EL2.FM0 == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AM0 == '1' && (IsVirtualSErrorPending() || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt; (PSTATE.EL);
```

## Library pseudocode for shared/functions/system/IsASEInstructionIsPhysicalSErrorPending

```
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.
// Returns TRUE if a physical SError interrupt is pending.
boolean IsASEInstruction(); IsPhysicalSErrorPending();
```

## Library pseudocode for shared/functions/system/IsCMOWControlledInstructionIsSErrorEdgeTriggered

```
// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CIGDVAC, or DC CIGVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsCMOWControlledInstruction(); IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;

    if ELUsingAArch32(target_el) then
        if syndrome<11:10> != '00' then
            // AArch32 and not Uncontainable.
            return TRUE;
    else
        if syndrome<24> == '0' && syndrome<5:0> != '000000' then
            // AArch64 and neither IMPLEMENTATION_DEFINED syndrome nor Uncategorized.
            return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

## Library pseudocode for shared/functions/system/IsCurrentSecurityStateIsSecure

```
// IsCurrentSecurityState()
// =====
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.
// IsSecure()
// =====
// Returns TRUE if current Exception level is in Secure state.

boolean IsCurrentSecurityState(IsSecure())
if SecurityStateHaveEL ss)
return( ) && !UsingAArch32() && PSTATE.EL == EL3 then
return TRUE;
elseif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
return TRUE;
return IsSecureBelowEL3CurrentSecurityStateEL3() == ss;();
```

## Library pseudocode for shared/functions/system/IsEventRegisterSetIsSecureBelowEL3

```
// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.
// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsEventRegisterSet()
return EventRegister == '1';IsSecureBelowEL3()
if HaveEL(EL3) then
return SCR_GEN[ ].NS == '0';
elseif HaveEL(EL2) && (!HaveSecureEL2Ext() || !HaveAArch64()) then
// If Secure EL2 is not an architecture option then we must be Non-secure.
return FALSE;
else
// TRUE if processor is Secure or FALSE if Non-secure.
return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

## Library pseudocode for shared/functions/system/IsHighestELIsSecureEL2Enabled

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsHighestEL(bits(2) el)
return IsSecureEL2Enabled()
if (EL2) && HaveSecureEL2Ext() then
if HaveEL(EL3) then
if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
return TRUE;
else
return FALSE;
else
return IsSecureHighestELHaveEL() == el;();
else
return FALSE;
```

## Library pseudocode for shared/functions/system/~~IsInHost~~~~IsSynchronizablePhysicalErrorPending~~

```
// IsInHost()
// =====
// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsInHost()
    returnIsSynchronizablePhysicalErrorPending(); ELIsInHost(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/~~IsPhysicalErrorPending~~~~IsVirtualErrorPending~~

```
// Returns TRUE if a physical SError interrupt is pending.
// Returns TRUE if a virtual SError interrupt is pending.
boolean IsPhysicalErrorPending(); IsVirtualErrorPending();
```

## Library pseudocode for shared/functions/system/~~IsSErrorEdgeTriggered~~~~LocalTimeoutEvent~~

```
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.
boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    ifLocalTimeoutEvent(integer localtimeout); HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;
    if ELUsingAArch32(target_el) then
        if syndrome<11:10> != '00' then
            // AArch32 and not Uncontainable.
            return TRUE;
    else
        if syndrome<24> == '0' && syndrome<5:0> != '000000' then
            // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor Uncategorized.
            return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

## Library pseudocode for shared/functions/system/~~IsSecure~~~~Mode\_Bits~~

```
// IsSecure()
// =====
// Returns TRUE if current Exception level is in Secure state.
boolean constant bits(5) IsSecure()
    ifM32_User == '10000';
    constant bits(5) HaveEL(M32_FIQ == '10001';
    constant bits(5) EL3) && !M32_IRQ == '10010';
    constant bits(5) UsingAArch32() && PSTATE.EL == M32_Svc == '10011';
    constant bits(5) EL3 then
        return TRUE;
    elseifM32_Monitor == '10110';
    constant bits(5) HaveEL(M32_Abort == '10111';
    constant bits(5) EL3) && M32_Hyp == '11010';
    constant bits(5) UsingAArch32() && PSTATE.M == M32_Undef == '11011';
    constant bits(5) M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3(); M32_System == '11111';
```

## Library pseudocode for shared/functions/system/**IsSecureBelowEL3NonSecureOnlyImplementation**

```
// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean IsSecureBelowEL3()
    if NonSecureOnlyImplementation()
        return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation"; HaveEL(EL3) then
            return SCR_GEN[].NS == '0';
        elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || !HaveAArch64()) then
            // If Secure EL2 is not an architecture option then we must be Non-secure.
            return FALSE;
        else
            // TRUE if processor is Secure or FALSE if Non-secure.
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

## Library pseudocode for shared/functions/system/**IsSecureEL2EnabledPLOfEL**

```
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.
// PLOfEL()
// =====

boolean PrivilegeLevel IsSecureEL2Enabled()
    if PLOfEL(bits(2) el)
        case el of
            when HaveELEL3 (return if !HaveAArch64() then PL1 else PL3;
            when EL2) && return HaveSecureEL2ExtPL2() then
                if;
            when HaveELEL1 (return EL3PL1) then
                if !;
            when ELUsingAArch32EL0 (return EL3PL0) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return SecureOnlyImplementation();
    else
        return FALSE;
```

## Library pseudocode for shared/functions/system/**IsSynchronizablePhysicalSErrorPendingPSTATE**

```
ProcState // Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending(); PSTATE;
```

## Library pseudocode for shared/functions/system/**IsVirtualSErrorPendingPhysicalCountInt**

```
// Returns TRUE if a virtual SError interrupt is pending.
boolean // PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

bits(64) IsVirtualSErrorPending(); PhysicalCountInt()
    return PhysicalCount<87:24>;
```

## Library pseudocode for shared/functions/system/LocalTimeoutEventPrivilegeLevel

```
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.  
boolean enumeration LocalTimeoutEvent(integer localtimeout); PrivilegeLevel {PL3, PL2, PL1, PL0};
```

## Library pseudocode for shared/functions/system/Mode\_BitsProcState

```
constant bits(5) type M32_User = '10000';  
constant bits(5) ProcState is (  
    bits (1) N, // Negative condition flag  
    bits (1) Z, // Zero condition flag  
    bits (1) C, // Carry condition flag  
    bits (1) V, // Overflow condition flag  
    bits (1) D, // Debug mask bit [AArch64 only]  
    bits (1) A, // SError interrupt mask bit  
    bits (1) I, // IRQ mask bit  
    bits (1) F, // FIQ mask bit  
    bits (1) PAN, // Privileged Access Never Bit [v8.1]  
    bits (1) UAO, // User Access Override [v8.2]  
    bits (1) DIT, // Data Independent Timing [v8.4]  
    bits (1) TCO, // Tag Check Override [v8.5, AArch64 only]  
    bits (2) BTYPE, // Branch Type [v8.5]  
    bits (1) ALLINT, // Interrupt mask bit  
    bits (1) SS, // Software step bit  
    bits (1) IL, // Illegal Execution state bit  
    bits (2) EL, // Exception level  
    bits (1) nRW, // not Register Width: 0=64, 1=32  
    bits (1) M32_FIQ = '10001';  
constant bits(5) M32_IRQ = '10010';  
constant bits(5) M32_Svc = '10011';  
constant bits(5) M32_Monitor = '10110';  
constant bits(5) M32_Abort = '10111';  
constant bits(5) M32_Hyp = '11010';  
constant bits(5) M32_Undef = '11011';  
constant bits(5) M32_System = '11111'; // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]  
    bits (1) Q, // Cumulative saturation flag [AArch32 only]  
    bits (4) GE, // Greater than or Equal flags [AArch32 only]  
    bits (1) SSBS, // Speculative Store Bypass Safe  
    bits (8) IT, // If-then bits, RES0 in CPSR [AArch32 only]  
    bits (1) J, // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]  
    bits (1) T, // T32 bit, RES0 in CPSR [AArch32 only]  
    bits (1) E, // Endianness bit [AArch32 only]  
    bits (5) M // Mode field [AArch32 only]  
)
```

## Library pseudocode for shared/functions/system/NonSecureOnlyImplementationRestoredITBits

```
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implementation.
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

booleanbits(8) NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IT is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IT == '1' then
        ifConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

## Library pseudocode for shared/functions/system/PLOfELSCRTYPE

```
// PLOfEL()
// =====
PrivilegeLeveltype PLOfEL(bits(2) el)
    case el of
        whenSCRTYPE; EL3 return if !HaveAArch64() then PL1 else PL3;
        when EL2 return PL2;
        when EL1 return PL1;
        when EL0 return PL0;
```

## Library pseudocode for shared/functions/system/PSTATESCR\_GEN

```
// SCR_GEN[]
// =====
SCRTYPE SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtendProcState PSTATE; {SCR};
    else
        r = SCR_EL3;
    return r;
```

## Library pseudocode for shared/functions/system/PhysicalCountIntSecureOnlyImplementation

```
// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.
// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this implementation.

bits(64)boolean PhysicalCountInt()
return PhysicalCount<87:24>;SecureOnlyImplementation()
return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

## Library pseudocode for shared/functions/system/PrivilegeLevelSecurityState

```
enumeration PrivilegeLevel {SecurityState {PL3, SS_NonSecure, PL2, SS_Secure
}; PL1, PL0};
```

## Library pseudocode for shared/functions/system/ProcStateSecurityStateAtEL

```
type// SecurityStateAtEL()
// =====
// Returns the effective security state at the exception level based off current settings.

SecurityState ProcState is (
    bits (1) N,          // Negative condition flag
    bits (1) Z,          // Zero condition flag
    bits (1) C,          // Carry condition flag
    bits (1) V,          // Overflow condition flag
    bits (1) D,          // Debug mask bit [AArch64 only]
    bits (1) A,          // SError interrupt mask bit
    bits (1) I,          // IRQ mask bit
    bits (1) F,          // FIQ mask bit
    bits (1) PAN,        // Privileged Access Never Bit [v8.1]
    bits (1) UAO,        // User Access Override [v8.2]
    bits (1) DIT,        // Data Independent Timing [v8.4]
    bits (1) TCO,        // Tag Check Override [v8.5, AArch64 only]
    bits (2) BTYPE,      // Branch Type [v8.5]
    bits (1) ALLINT,     // Interrupt mask bit
    bits (1) SS,         // Software step bit
    bits (1) IL,         // Illegal Execution state bit
    bits (2) EL,         // Exception level
    bits (1) nRW,        // not Register Width: 0=64, 1=32
    bits (1)SecurityStateAtEL(bits(2)-EL)
    if ! (EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
    elsif EL == EL3 then
        return SS_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR.NS == '1' then SS_NonSecure else SS_SecureSPHaveEL, // Stack pointer sel
    bits (1) Q,          // Cumulative saturation flag [AArch32 only]
    bits (4) GE,         // Greater than or Equal flags [AArch32 only]
    bits (1) SSBS,       // Speculative Store Bypass Safe
    bits (8) IT,         // If-then bits, RES0 in CPSR [AArch32 only]
    bits (1) J,          // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,          // T32 bit, RES0 in CPSR [AArch32 only]
    bits (1) E,          // Endianness bit [AArch32 only]
    bits (5) M,          // Mode field [AArch32 only]
);
```

## Library pseudocode for shared/functions/system/RestoredITBitsSendEvent

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) // Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed. RestoredITBits(bits(N)
    it = spsr<15:10,26:25>;

    // When PSTATE.IT is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IT == '1' then
        if SendEvent(); ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

## Library pseudocode for shared/functions/system/SCRTTypeSendEventLocal

```
type // SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed. SCRTType; SendEventLocal
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/system/SCR\_GENSetAccumulatedFPEExceptions

```
// SCR_GEN[]
// =====
SCRType // Stores FP Exceptions accumulated by the PE. SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assertSetAccumulatedFPEExceptions(bits(8) accumulated_exceptions); HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SCR);
    else
        r = SCR_EL3;
    return r;
```

```

// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this implementation.

boolean// SetPSTATEFromPSR()
// ===== SecureOnlyImplementation()
return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";SetPSTATEFromPSR(bits(N) spsr)
boolean illegal_psr_state = IllegalExceptionReturn(spsr);
SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
bits(N) spsr = spsr_in;
boolean from_aarch64 = !UsingAArch32();
assert N == (if from_aarch64 then 64 else 32);
PSTATE.SS = DebugExceptionReturnSS(spsr);
ShouldAdvanceSS = FALSE;
if illegal_psr_state then
    PSTATE.IL = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
    if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
    if HaveDITEExt() then PSTATE.DIT = bit UNKNOWN;
    if HaveMTEEExt() then PSTATE.TCO = bit UNKNOWN;
else
    // State that is reinstated only on a legal exception return
    PSTATE.IL = spsr<20>;
    if spsr<4> == '1' then // AArch32 state
        AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
        if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
    else // AArch64 state
        PSTATE.nRW = '0';
        PSTATE.EL = spsr<3:2>;
        PSTATE.SP = spsr<0>;
        if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
        if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
        if HaveUAOExt() then PSTATE.UAO = spsr<23>;
        if HaveDITEExt() then PSTATE.DIT = spsr<24>;
        if HaveMTEEExt() then PSTATE.TCO = spsr<25>;

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then // AArch32 state
        PSTATE.Q = spsr<27>;
        PSTATE.IT = RestoredITBits(spsr);
        ShouldAdvanceIT = FALSE;
        if HaveDITEExt() then PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
        PSTATE.GE = spsr<19:16>;
        PSTATE.E = spsr<9>;
        PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
        PSTATE.T = spsr<5>; // PSTATE.J is RES0
    else // AArch64 state
        if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>;
        PSTATE.<D,A,I,F> = spsr<9:6>; // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;

```

### Library pseudocode for shared/functions/system/~~SecurityStateShouldAdvanceIT~~

```
enumeration boolean ShouldAdvanceIT; SecurityState {  
    SS_NonSecure,  
    SS_Secure  
};
```

### Library pseudocode for shared/functions/system/~~SecurityStateAtELShouldAdvanceSS~~

```
// SecurityStateAtEL()  
// =====  
// Returns the effective security state at the exception level based off current settings.  
  
SecurityState boolean ShouldAdvanceSS; SecurityStateAtEL(bits(2) EL)  
    if !HaveEL(EL3) then  
        if SecureOnlyImplementation() then  
            return SS_Secure;  
        else  
            return SS_NonSecure;  
    elseif EL == EL3 then  
        return SS_Secure;  
    else  
        // For EL2 call only when EL2 is enabled in current security state  
        assert(EL != EL2 || EL2Enabled());  
        if !ELUsingAArch32(EL3) then  
            return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;  
        else  
            return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
```

### Library pseudocode for shared/functions/system/~~SendEventSpeculationBarrier~~

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.  
// When a PE executes the SEV instruction, it causes this function to be executed.  
SendEvent();SpeculationBarrier();
```

### Library pseudocode for shared/functions/system/~~SendEventLocalSynchronizeContext~~

```
// SendEventLocal()  
// =====  
// Set the local Event Register of this PE.  
// When a PE executes the SEVL instruction, it causes this function to be executed.  
  
SendEventLocal()  
    EventRegister = '1';  
    return;SynchronizeContext();
```

### Library pseudocode for shared/functions/system/~~SetAccumulatedFPExceptionsSynchronizeErrors~~

```
// Stores FP Exceptions accumulated by the PE. // Implements the error synchronization event.  
SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);SynchronizeErrors();
```

## Library pseudocode for shared/functions/ system/~~SetPSTATEFromPSR~~~~TakeUnmaskedPhysicalSErrorInterrupts~~

```
// SetPSTATEFromPSR()
// =====// Take any pending unmasked physical SError interrupt.

SetPSTATEFromPSR(bits(N) spsr)
    boolean illegal_psr_state = TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req); IllegalExceptionRe
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSEExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIEExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUA0Ext() then PSTATE.UA0 = bit UNKNOWN;
        if HaveDITEExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32 state
            AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
            if HaveSSBSEExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
            if HaveBTIEExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSEExt() then PSTATE.SSBS = spsr<12>;
            if HaveUA0Ext() then PSTATE.UA0 = spsr<23>;
            if HaveDITEExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEEExt() then PSTATE.TCO = spsr<25>;

        // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
        // copied from SPSR.
        if PSTATE.IL == '1' && PSTATE.nRW == '1' then
            if ConstrainUnpredictableBool(Unpredictable\_ILZEROT) then spsr<5> = '0';

        // State that is reinstated regardless of illegal exception return
        PSTATE.<N,Z,C,V> = spsr<31:28>;
        if HavePANExt() then PSTATE.PAN = spsr<22>;
        if PSTATE.nRW == '1' then // AArch32 state
            PSTATE.Q = spsr<27>;
            PSTATE.IT = RestoredITBits(spsr);
            ShouldAdvanceIT = FALSE;
            if HaveDITEExt() then PSTATE.DIT = (if (Restarting()) || from_aarch64) then spsr<24> else spsr<21>;
            PSTATE.GE = spsr<19:16>;
            PSTATE.E = spsr<9>;
            PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
            PSTATE.T = spsr<5>; // PSTATE.J is RES0
        else // AArch64 state
            if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>;
            PSTATE.<D,A,I,F> = spsr<9:6>; // No PSTATE.<Q,IT,GE,E,T> in AArch64 state

    return;
```

**Library pseudocode for shared/functions/  
system/ShouldAdvanceITTakeUnmaskedSErrorInterrupts**

```
boolean ShouldAdvanceIT; // Take any pending unmasked physical SError interrupt or unmasked virtual SError  
// interrupt.TakeUnmaskedSErrorInterrupts();
```

**Library pseudocode for shared/functions/system/ShouldAdvanceSSThisInstr**

```
boolean ShouldAdvanceSS; bits(32) ThisInstr();
```

**Library pseudocode for shared/functions/system/SpeculationBarrierThisInstrLength**

```
integer SpeculationBarrier(); ThisInstrLength();
```

**Library pseudocode for shared/functions/system/SynchronizeContextUnreachable**

```
SynchronizeContext(); Unreachable()  
assert FALSE;
```

**Library pseudocode for shared/functions/system/SynchronizeErrorsUsingAArch32**

```
// Implements the error synchronization event. // UsingAArch32()  
// =====  
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.  
  
boolean  
SynchronizeErrors(); UsingAArch32()  
boolean aarch32 = (PSTATE.nRW == '1');  
if !HaveAArch32() then assert !aarch32;  
if !HaveAArch64() then assert aarch32;  
return aarch32;
```

**Library pseudocode for shared/functions/  
system/TakeUnmaskedPhysicalSErrorInterruptsVirtualFIQPending**

```
// Take any pending unmasked physical SError interrupt. // Returns TRUE if there is any pending virtual FIQ  
boolean  
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req); VirtualFIQPending();
```

**Library pseudocode for shared/functions/  
system/TakeUnmaskedSErrorInterruptsVirtualIRQPending**

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError  
// interrupt. // Returns TRUE if there is any pending virtual IRQ.  
boolean  
TakeUnmaskedSErrorInterrupts(); VirtualIRQPending();
```

**Library pseudocode for shared/functions/system/ThisInstrWFXType**

```
bits(32) enumeration ThisInstr(); WFXType {WFXType_WFE, WFXType_WFI, WFXType_WFET, WFXType_WFIT};
```

## Library pseudocode for shared/functions/system/**ThisInstrLengthWaitForEvent**

```
integer// WaitForEvent()  
// =====  
// PE optionally suspends execution until one of the following occurs:  
// - A WFE wake-up event.  
// - A reset.  
// - The implementation chooses to resume execution.  
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs  
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of  
// suspension causes the Event Register to be cleared. ThisInstrLength();WaitForEvent(integer localtimeou  
if !(IsEventRegisterSet() || (HaveFeatWfXT() && LocalTimeoutEvent(localtimeout))) then  
    EnterLowPowerState();  
return;
```

## Library pseudocode for shared/functions/system/**UnreachableWaitForInterrupt**

```
// WaitForInterrupt()  
// =====  
// PE optionally suspends execution until one of the following occurs:  
// - A WFI wake-up event.  
// - A reset.  
// - The implementation chooses to resume execution.  
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.  
  
WaitForInterrupt(integer localtimeout)  
if !(HaveFeatWfXT() && LocalTimeoutEvent(localtimeout)) then  
    EnterLowPowerStateUnreachable()  
assert FALSE;()  
return;
```

Library pseudocode for shared/  
functions/systemunpredictable/UsingAArch32ConstrainUnpredictable

```

// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.
// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

boolean// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !ConstrainUnpredictable(HaveAArch32Unpredictable() then assert !aarch32;
    if !which)
        case which of
            when
                return Constraint_UNDEF;
            when Unpredictable_WBOVERLAPLD
                return Constraint_WBSUPPRESS; // return loaded value
            when Unpredictable_WBOVERLAPST
                return Constraint_NONE; // store pre-writeback value
            when Unpredictable_LDPOVERLAP
                return Constraint_UNDEF; // instruction is UNDEFINED
            when Unpredictable_BASEOVERLAP
                return Constraint_UNKNOWN; // use UNKNOWN address
            when Unpredictable_DATAOVERLAP
                return Constraint_UNKNOWN; // store UNKNOWN value
            when Unpredictable_DEVPAGE2
                return Constraint_FAULT; // take an alignment fault
            when Unpredictable_DEVICETAGSTORE
                return Constraint_NONE; // Do not take a fault
            when Unpredictable_INSTRDEVICE
                return Constraint_NONE; // Do not take a fault
            when Unpredictable_RESCPACR
                return Constraint_TRUE; // Map to UNKNOWN value
            when Unpredictable_RESMAIR
                return Constraint_UNKNOWN; // Map to UNKNOWN value
            when Unpredictable_S1CTAGGED
                return Constraint_FALSE; // SCTL_R_ELx.C == '0' marks address as untagged
            when Unpredictable_S2RESMEMATTR
                return Constraint_NC; // Map to Noncacheable value
            when Unpredictable_RESTEXCB
                return Constraint_UNKNOWN; // Map to UNKNOWN value
            when Unpredictable_RESDACR
                return Constraint_UNKNOWN; // Map to UNKNOWN value
            when Unpredictable_RESPRRR
                return Constraint_UNKNOWN; // Map to UNKNOWN value
            when Unpredictable_RESVTCRS
                return Constraint_UNKNOWN; // Map to UNKNOWN value
            when Unpredictable_RESTnSZ
                return Constraint_FORCE; // Map to the limit value
            when Unpredictable_OORTnSZ
                return Constraint_FORCE; // Map to the limit value
            when Unpredictable_LARGEIPA
                return Constraint_FORCE; // Restrict the IA size to the PAMax value
            when Unpredictable_ESRCONDPASS
                return Constraint_FALSE; // Report as "AL"
            when Unpredictable_ILZEROIT
                return Constraint_FALSE; // Do not zero PSTATE.IT
            when Unpredictable_ILZEROT
                return Constraint_FALSE; // Do not zero PSTATE.T
            when Unpredictable_BPVECTORCATCHPRI
                return Constraint_TRUE; // Debug Vector Catch: match on 2nd halfword
            when Unpredictable_VCMATCHHALF
                return Constraint_FALSE; // No match

```

```

when Unpredictable_VCMATCHDAPA
    return Constraint_FALSE; // No match on Data Abort or Prefetch abort
when Unpredictable_WPMASKANDBAS
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_WPBASCONTIGUOUS
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESWPMASK
    return Constraint_DISABLED; // Watchpoint disabled
when Unpredictable_WPMASKEDBITS
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESBPWCTRL
    return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESBPTYPE
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMATCHHALF
    return Constraint_FALSE; // No match
when Unpredictable_BPMISMATCHHALF
    return Constraint_FALSE; // No match
when Unpredictable_RESTARTALIGNPC
    return Constraint_FALSE; // Do not force alignment
when Unpredictable_RESTARTZEROUPPERPC
    return Constraint_TRUE; // Force zero extension
when Unpredictable_ZEROUPPER
    return Constraint_TRUE; // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC
    return Constraint_TRUE; // zero top half of PC
when Unpredictable_A32FORCEALIGNPC
    return Constraint_FALSE; // Do not force alignment
when Unpredictable_SMD
    return Constraint_UNDEF; // disabled SMC is Unallocated
when Unpredictable_NVNV1
    return Constraint_NVNV1_00; // Map unpredictable configuration of HCR_EL2<NV,NV1>
                                // to NV = 0 and NV1 = 0
when Unpredictable_Shareability
    return Constraint_OSH; // Map reserved encoding of shareability to outer shareable
when Unpredictable_AFUPDATE
    return Constraint_TRUE; // AF update for alignment or permission fault
when Unpredictable_IESBinDebug
    return Constraint_TRUE; // Use SCTLR[.IESB] in Debug state
when Unpredictable_BADPMSFCR
    return Constraint_TRUE; // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
when Unpredictable_ZEROBTYP
    return Constraint_TRUE; // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable_CLEARERRITEZERO
    return Constraint_FALSE; // Clearing sticky errors when instruction in flight
when Unpredictable_ALUEXCEPTIONRETURN
    return Constraint_UNDEF;
when Unpredictable_DBGxVR_RESS
    return Constraint_FALSE;
when Unpredictable_PMSCR_PCT
    return Constraint_PMSCR_PCT_VIRT;
when Unpredictable_WFXTDEBUG
    return Constraint_FALSE; // WFXt in Debug state does not execute as a NOP
when Unpredictable_LS64UNSUPPORTED
    return Constraint_LIMITED_ATOMICITY; // Accesses are not single-copy atomic above the byte level
// Misaligned exclusives, atomics, acquire/release to region that is not Normal Cacheable WB are a
when Unpredictable_MISALIGNEDATOMIC
    return Constraint_FALSE;

when Unpredictable_IGNORETRAPINDEBUG
    return Constraint_TRUE; // Trap to register access in debug state is ignored
when Unpredictable_PMUEVENTCOUNTER
    return Constraint_UNDEF;
return aarch32; // HaveAArch64Unpredictable_VMSR() then assert aarch32;
// Accesses to the register are UNDEFINED

```

## Library pseudocode for shared/ functions/systemunpredictable/VirtualFIQPendingConstrainUnpredictableBits

```
// Returns TRUE if there is any pending virtual FIQ.
boolean// ConstrainUnpredictableBits()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint,bits(width)) VirtualFIQPending();ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, Zeros(width)); // See notes; this is an example implementation only
    elsif c == Constraint_PMSCR_PCT_VIRT then
        return (c,Zeros(width));
    else
        return (c, bits(width) UNKNOWN); // bits result not used
```

## Library pseudocode for shared/ functions/systemunpredictable/VirtualIRQPendingConstrainUnpredictableBool

```
// Returns TRUE if there is any pending virtual IRQ.
// ConstrainUnpredictableBool()
// =====

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean VirtualIRQPending();ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

## Library pseudocode for shared/ functions/systemunpredictable/WFxTypeConstrainUnpredictableInteger

```
enumeration// ConstrainUnpredictableInteger()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint, integer) WFxType {ConstrainUnpredictableInteger(integer low, integer high, WFxType_WFE, which)

    c = WFxType_WFI, (which);

    if c == WFxType_WFET, WFxType_WFIT}; then
        return (c, low); // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN); // integer result not used
```

## Library pseudocode for shared/functions/systemunpredictable/WaitForEventConstraint

```
// WaitForEvent()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.enumeration

WaitForEvent(integer localtimeout)
    if !(Constraint { // GeneralIsEventRegisterSet() || (Constraint_NONE, // Instruction e
        // no change or side-effect to its described b
        Constraint_UNDEFEL0, // Instruction is UNDEFINED at EL0 only
        Constraint_NOP, // Instruction executes as NOP
        Constraint_TRUE,
        Constraint_FALSE,
        Constraint_DISABLED,
        Constraint_UNCOND, // Instruction executes unconditionally
        Constraint_COND, // Instruction executes conditionally
        Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
        // Load-store
        Constraint_WBSUPPRESS,
        Constraint_FAULT,
        Constraint_LIMITED_ATOMICITY, // Accesses are not single-copy atomic above the
        Constraint_NVNV1_00,
        Constraint_NVNV1_01,
        Constraint_NVNV1_11,
        Constraint_OSH, // Constrain to Outer shareable
        Constraint_ISH, // Constrain to Inner shareable
        Constraint_NSH, // Constrain to Nonshareable

        Constraint_NC, // Constrain to Noncacheable
        Constraint_WT, // Constrain to Writethrough
        Constraint_WB, // Constrain to Writeback

        // IPA too large
        Constraint_FORCE, Constraint_FORCENOSLCHECK,
        // PMSCR_PCT reserved values select Virtual timestamp
        EnterLowPowerState();
    return; Constraint_PMSCR_PCT_VIRT};
```



```

// WaitForInterrupt()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.enumeration

WaitForInterrupt(integer localtimeout)
    if !(Unpredictable { // VMSR on MVFRHaveFeatWFXT() &&Unpredictable_VMSR,
        // Writeback/transfer register overlap (load) LocalTimeoutEvent(localtimeout),
        // Writeback/transfer register overlap (store)
        Unpredictable_WBOVERLAPST,
        // Load Pair transfer register overlap
        Unpredictable_LDPOVERLAP,
        // Store-exclusive base/status register overlap
        Unpredictable_BASEOVERLAP,
        // Store-exclusive data/status register overlap
        Unpredictable_DATAOVERLAP,
        // Load-store alignment checks
        Unpredictable_DEVPAGE2,
        // Instruction fetch from Device memory
        Unpredictable_INSTRDEVICE,
        // Reserved CPACR value
        Unpredictable_RESCPACR,
        // Reserved MAIR value
        Unpredictable_RESMAIR,
        // Effect of SCTLr_ELx.C on Tagged attribute
        Unpredictable_S1CTAGGED,
        // Reserved Stage 2 MemAttr value
        Unpredictable_S2RESMEMATTR,
        // Reserved TEX:C:B value
        Unpredictable_RESTEXCB,
        // Reserved PRRR value
        Unpredictable_RESPPRRR,
        // Reserved DACR field
        Unpredictable_RESDACR,
        // Reserved VTCR.S value
        Unpredictable_RESVTCRS,
        // Reserved TCR.TnSZ value
        Unpredictable_RESTnSZ,
        // Reserved SCTLr_ELx.TCF value
        Unpredictable_RESTCF,
        // Tag stored to Device memory
        Unpredictable_DEVICETAGSTORE,
        // Out-of-range TCR.TnSZ value
        Unpredictable_0ORTnSZ,
        // IPA size exceeds PA size
        Unpredictable_LARGEIPA,
        // Syndrome for a known-passing conditional A32 instruction
        Unpredictable_ESRCONDPASS,
        // Illegal State exception: zero PSTATE.IT
        Unpredictable_ILZEROIT,
        // Illegal State exception: zero PSTATE.T
        Unpredictable_ILZEROT,
        // Debug: prioritization of Vector Catch
        Unpredictable_BPVECTORCATCHPRI,
        // Debug Vector Catch: match on 2nd halfword
        Unpredictable_VCMATCHHALF,
        // Debug Vector Catch: match on Data Abort or Prefetch abort
        Unpredictable_VCMATCHDAPA,
        // Debug watchpoints: non-zero MASK and non-ones BAS
        Unpredictable_WPMASKANDBAS,
        // Debug watchpoints: non-contiguous BAS
        Unpredictable_WPBASCONTIGUOUS,
        // Debug watchpoints: reserved MASK
        Unpredictable_RESWPMASK,
        // Debug watchpoints: non-zero MASKed bits of address
        Unpredictable_WPMASKEDBITS,
        // Debug breakpoints and watchpoints: reserved control bits
    })

```

```

Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZERoupperPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZERoupper,
// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZERoupperPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[.IESB] in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTtype,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// WFET or WFIT instruction in Debug state
Unpredictable_WFXTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Misaligned exclusives, atomics, acquire/release to region that is not Normal
Unpredictable_MISALIGNEDATOMIC,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGxVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behaviour.
EnterLowPowerState();
return;Unpredictable_PMSCR_PCT,
};

```

Library pseudocode for shared/  
functions/**unpredictablevector/ConstrainUnpredictableAdvSIMDExpandImm**

```

// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)
// AdvSIMDExpandImm()
// =====

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraintbits(64) ConstrainUnpredictable(AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8),
bits(64) imm64;
case cmode<3:1> of
when '000'
imm64 = UnpredictableReplicate which)
case which of
when( Unpredictable_VMSRZeros
return(24):imm8, 2);
when '001'
imm64 = Constraint_UNDEFReplicate;
when( Unpredictable_WBOVERLAPLDZeros
return(16):imm8: Constraint_WBSUPPRESSZeros; // return loaded value
when(8), 2);
when '010'
imm64 = Unpredictable_WBOVERLAPSTReplicate
return( Constraint_NONEZeros; // store pre-writeback value
when(8):imm8: Unpredictable_LDPOVERLAPZeros
return(16), 2);
when '011'
imm64 = Constraint_UNDEFReplicate; // instruction is UNDEFINED
when(imm8: Unpredictable_BASEOVERLAPZeros
return(24), 2);
when '100'
imm64 = Constraint_UNKNOWNReplicate; // use UNKNOWN address
when( Unpredictable_DATAOVERLAPZeros
return(8):imm8, 4);
when '101'
imm64 = Constraint_UNKNOWNReplicate; // store UNKNOWN value
when(imm8: Unpredictable_DEVPAGE2Zeros
return(8), 4);
when '110'
if cmode<0> == '0' then
imm64 = Constraint_FAULTReplicate; // take an alignment fault
when( Unpredictable_DEVICETAGSTOREZeros
return(16):imm8: Constraint_NONEOnes; // Do not take a fault
when(8), 2);
else
imm64 = Unpredictable_INSTRDEVICEReplicate
return( Constraint_NONEZeros; // Do not take a fault
when(8):imm8: Unpredictable_RESCPACR0nes
return(16), 2);
when '111'
if cmode<0> == '0' && op == '0' then
imm64 = Constraint_TRUEReplicate; // Map to UNKNOWN value
when(imm8, 8);
if cmode<0> == '0' && op == '1' then
imm8a = Unpredictable_RESMAIRReplicate
return(imm8<7>, 8); imm8b = Constraint_UNKNOWNReplicate; // Map to UNKNOWN value
when(imm8<6>, 8);
imm8c = Unpredictable_S1CTAGGEDReplicate
return(imm8<5>, 8); imm8d = Constraint_FALSEReplicate; // SCTLR_ELx.C == '0' marks address
when(imm8<4>, 8);
imm8e = Unpredictable_S2RESMEMATTRReplicate
return(imm8<3>, 8); imm8f = Constraint_NCREplicate; // Map to Noncacheable value
when(imm8<2>, 8);
imm8g = Unpredictable_RESTEXCBReplicate

```

```

return(imm8<1>, 8); imm8h = Constraint_UNKOWNReplicate; // Map to UNKOWN value
when(imm8<0>, 8);
    imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
    if cmode<0> == '1' && op == '0' then
        imm32 = imm8<7>:NOT(imm8<6>): Unpredictable_RESDACRReplicate
    return(imm8<6>,5):imm8<5:0>: Constraint_UNKOWNZeros; // Map to UNKOWN value
when(19);
    imm64 = Unpredictable_RESPPRRReplicate
    return(imm32, 2);
    if cmode<0> == '1' && op == '1' then
        if Constraint_UNKOWNUsingAArch32; // Map to UNKOWN value
    when() then ReservedEncoding();
        imm64 = imm8<7>:NOT(imm8<6>): Unpredictable_RESVTCRSReplicate
    return(imm8<6>,8):imm8<5:0>: Constraint_UNKOWNZeros; // Map to UNKOWN value
when Unpredictable_RESTnSZ
    return Constraint_FORCE; // Map to the limit value
when Unpredictable_OORTnSZ
    return Constraint_FORCE; // Map to the limit value
when Unpredictable_LARGEIPA
    return Constraint_FORCE; // Restrict the IA size to the PAMax value
when Unpredictable_ESRCONDPASS
    return Constraint_FALSE; // Report as "AL"
when Unpredictable_ILZEROIT
    return Constraint_FALSE; // Do not zero PSTATE.IT
when Unpredictable_ILZEROT
    return Constraint_FALSE; // Do not zero PSTATE.T
when Unpredictable_BPVECTORCATCHPRI
    return Constraint_TRUE; // Debug Vector Catch: match on 2nd halfword
when Unpredictable_VCMATCHHALF
    return Constraint_FALSE; // No match
when Unpredictable_VCMATCHDAPA
    return Constraint_FALSE; // No match on Data Abort or Prefetch abort
when Unpredictable_WPMASKANDBAS
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_WPBASCONTIGUOUS
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESWPMASK
    return Constraint_DISABLED; // Watchpoint disabled
when Unpredictable_WPMASKEDBITS
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESBPWPCTRL
    return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESBPTYPE
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMATCHHALF
    return Constraint_FALSE; // No match
when Unpredictable_BPMISMATCHHALF
    return Constraint_FALSE; // No match
when Unpredictable_RESTARTALIGNPC
    return Constraint_FALSE; // Do not force alignment
when Unpredictable_RESTARTZEROUPPERPC
    return Constraint_TRUE; // Force zero extension
when Unpredictable_ZEROUPPER
    return Constraint_TRUE; // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC
    return Constraint_TRUE; // zero top half of PC
when Unpredictable_A32FORCEALIGNPC
    return Constraint_FALSE; // Do not force alignment
when Unpredictable_SMD
    return Constraint_UNDEF; // disabled SMC is Unallocated
when Unpredictable_NVNV1
    return Constraint_NVNV1_00; // Map unpredictable configuration of HCR_EL2<NV,NV1>
    // to NV = 0 and NV1 = 0
when Unpredictable_Shareability
    return Constraint_OSH; // Map reserved encoding of shareability to Outer Shareable
when Unpredictable_AFUPDATE
    // AF update for alignment or Permission fault

```

```

        return Constraint_TRUE;
    when Unpredictable_IESBinDebug // Use SCTLR[].IESB in Debug state
        return Constraint_TRUE;
    when Unpredictable_BADPMSFCR // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
        return Constraint_TRUE;
    when Unpredictable_ZEROBTYPEx
        return Constraint_TRUE; // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
    when Unpredictable_CLEARERRITEZERO // Clearing sticky errors when instruction in flight
        return Constraint_FALSE;
    when Unpredictable_ALUEXCEPTIONRETURN
        return Constraint_UNDEF;
    when Unpredictable_DBGxVR_RESS
        return Constraint_FALSE;
    when Unpredictable_PMSCR_PCT
        return Constraint_PMSCR_PCT_VIRT;
    when Unpredictable_WFXTDEBUG
        return Constraint_FALSE; // WFXt in Debug state does not execute as a NOP
    when Unpredictable_LS64UNSUPPORTED
        return Constraint_LIMITED_ATOMICITY; // Accesses are not single-copy atomic above the byte level
    // Misaligned exclusives, atomics, acquire/release to region that is not Normal Cacheable WB are a
    when Unpredictable_MISALIGNEDATOMIC
        return Constraint_FALSE;

    when Unpredictable_IGNORETRAPINDEBUG
        return Constraint_TRUE; // Trap to register access in debug state is ignored
    when Unpredictable_PMUEVENTCOUNTER
        return Constraint_UNDEF; // Accesses to the register are UNDEFINED(48);

return imm64;

```

## Library pseudocode for shared/ functions/~~unpredictablevector~~/ConstrainUnpredictableBitsMatMulAdd

```
// ConstrainUnpredictableBits()
// =====
// MatMulAdd()
// =====
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint, bits(width)) bits(N) ConstrainUnpredictableBits(MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
    assert N == 128;

    bits(N) result;
    bits(32) sum;
    integer prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = UnpredictableElem which)

    c = [addend, 2*i + j, 32];
    for k = 0 to 7
        prod = ConstrainUnpredictableInt(which);

    if c == ( Constraint_UNKNOWNElem then
        return (c, [op1, 8*i + k, 8], op1_unsigned) * ZerosInt(width)); // See notes; this is an
    elsif c == ( Constraint_PMSCR_PCT_VIRTElem then
        return (c, [op2, 8*j + k, 8], op2_unsigned);
        sum = sum + prod; ZerosElem(width));
    else
        return (c, bits(width) UNKNOWN); // bits result not used [result, 2*i + j, 32] = sum;

    return result;
```

## Library pseudocode for shared/ functions/~~unpredictablevector~~/ConstrainUnpredictableBoolPolynomialMult

```
// ConstrainUnpredictableBool()
// =====
// PolynomialMult()
// =====

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

booleanbits(M+N) ConstrainUnpredictableBool(PolynomialMult(bits(M) op1, bits(N) op2)
  result =UnpredictableZeros which)

  c =(M+N);
  extended_op2 = ConstrainUnpredictableZeroExtend(which);
  assert c IN {(op2, M+N);
  for i=0 to M-1
    if op1<i> == '1' then
      result = result EORConstrain_TRUELSL, Constrain_FALSE};
  return (c == Constrain_TRUE);(extended_op2, i);
  return result;
```

## Library pseudocode for shared/ functions/~~unpredictablevector~~/ConstrainUnpredictableIntegerSatQ

```
// ConstrainUnpredictableInteger()
// =====
// SatQ()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer)(bits(N), boolean) ConstrainUnpredictableInteger(integer low, integer high,SatQ(integ
  (result, sat) = if unsigned then UnpredictableUnsignedSatQ which)

  c =(i, N) else ConstrainUnpredictableSignedSatQ(which);

  if c == Constraint_UNKNOWN then
    return (c, low); // See notes; this is an example implementation only
  else
    return (c, integer UNKNOWN); // integer result not used(i, N);
  return (result, sat);
```

```

enumeration// SignedSatQ()
// =====
(bits(N), boolean) Constraint {// GeneralSignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1; saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1)); saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
    Constraint_NONE, // Instruction executes with
                    // no change or side-effect to its described b
    Constraint_UNKNOWN, // Destination register has UNKNOWN value
    Constraint_UNDEF, // Instruction is UNDEFINED
    Constraint_UNDEFEL0, // Instruction is UNDEFINED at EL0 only
    Constraint_NOP, // Instruction executes as NOP
    Constraint_TRUE,
    Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND, // Instruction executes unconditionally
    Constraint_COND, // Instruction executes conditionally
    Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
    // Load-store
    Constraint_WBSUPPRESS,
    Constraint_FAULT,
    Constraint_LIMITED_ATOMICITY, // Accesses are not single-copy atomic above the
    Constraint_NVNV1_00,
    Constraint_NVNV1_01,
    Constraint_NVNV1_11,
    Constraint_OSH, // Constrain to Outer Shareable
    Constraint_ISH, // Constrain to Inner Shareable
    Constraint_NSH, // Constrain to Nonshareable
    Constraint_NC, // Constrain to Noncacheable
    Constraint_WT, // Constrain to Writethrough
    Constraint_WB, // Constrain to Writeback
    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLCHECK,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT};

```

Library pseudocode for shared/  
functions/**unpredictablevector/UnpredictableUnsignedRSqrtEstimate**

```

enumeration // UnsignedRSqrtEstimate()
// =====
bits(N) Unpredictable { // VMSR on MVFR UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result =
            Unpredictable_VMSR,
            // Writeback/transfer register overlap (load)(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate =
            Unpredictable_WBOVERLAPLD,
            // Writeback/transfer register overlap (store){
            Unpredictable_WBOVERLAPST,
            // Load Pair transfer register overlap(operand<31:23>), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> ;
        Unpredictable_LDPOVERLAP,
        // Store-exclusive base/status register overlap
        Unpredictable_BASEOVERLAP,
        // Store-exclusive data/status register overlap
        Unpredictable_DATAOVERLAP,
        // Load-store alignment checks
        Unpredictable_DEVPAGE2,
        // Instruction fetch from Device memory
        Unpredictable_INSTRDEVICE,
        // Reserved CPACR value
        Unpredictable_RESCPACR,
        // Reserved MAIR value
        Unpredictable_RESMAIR,
        // Effect of SCTLR_ELx.C on Tagged attribute
        Unpredictable_S1CTAGGED,
        // Reserved Stage 2 MemAttr value
        Unpredictable_S2RESMEMATTR,
        // Reserved TEX:C:B value
        Unpredictable_RESTEXCB,
        // Reserved PRRR value
        Unpredictable_RESPRRR,
        // Reserved DACR field
        Unpredictable_RESDACR,
        // Reserved VTCR.S value
        Unpredictable_RESVTCRS,
        // Reserved TCR.TnSZ value
        Unpredictable_RESTnSZ,
        // Reserved SCTLR_ELx.TCF value
        Unpredictable_RESTCF,
        // Tag stored to Device memory
        Unpredictable_DEVICETAGSTORE,
        // Out-of-range TCR.TnSZ value
        Unpredictable_OORTnSZ,
        // IPA size exceeds PA size
        Unpredictable_LARGEIPA,
        // Syndrome for a known-passing conditional A32 instruction
        Unpredictable_ESRCONDPASS,
        // Illegal State exception: zero PSTATE.IT
        Unpredictable_ILZEROIT,
        // Illegal State exception: zero PSTATE.T
        Unpredictable_ILZEROT,
        // Debug: prioritization of Vector Catch
        Unpredictable_BPVECTORCATCHPRI,
        // Debug Vector Catch: match on 2nd halfword
        Unpredictable_VCMATCHHALF,
        // Debug Vector Catch: match on Data Abort or Prefetch abort
        Unpredictable_VCMATCHDAPA,
        // Debug watchpoints: non-zero MASK and non-ones BAS
        Unpredictable_WPMASKANDBAS,

```

```

// Debug watchpoints: non-contiguous BAS
Unpredictable_WPBASCONTIGUOUS,
// Debug watchpoints: reserved MASK
Unpredictable_RESWPMASK,
// Debug watchpoints: non-zero MASKed bits of address
Unpredictable_WPMASKEDBITS,
// Debug breakpoints and watchpoints: reserved control bits
Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[.IESB] in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// WFET or WFIT instruction in Debug state
Unpredictable_WFXTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Misaligned exclusives, atomics, acquire/release to region that is not Normal
Unpredictable_MISALIGNEDATOMIC,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGxVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behaviour.
Unpredictable_PMSCR_PCT,
};(N-9);

return result;

```

```

// AdvSIMDExpandImm()
// =====
// UnsignedRecipEstimate()
// =====

bits(64)bits(N) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
  bits(64) imm64;
  case cmode<3:1> of
    when '000'
      imm64 = UnsignedRecipEstimate(bits(N) operand);
  assert N == 32;
  bits(N) result;
  if operand<N-1> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Replicate(Zeros(24):imm8, 2);
  when '001'
    imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
  when '010'
    imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
  when '011'
    imm64 = Replicate(imm8:Zeros(24), 2);
  when '100'
    imm64 = Replicate(Zeros(8):imm8, 4);
  when '101'
    imm64 = Replicate(imm8:Zeros(8), 4);
  when '110'
    if cmode<0> == '0' then
      imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
    else
      imm64 = (N);
  else
    // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

    // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
    increasedprecision = FALSE;
    estimate = ReplicateRecipEstimate(ZerosUInt(8):imm8:Ones(16), 2);
  when '111'
    if cmode<0> == '0' && op == '0' then
      imm64 = Replicate(imm8, 8);
    if cmode<0> == '0' && op == '1' then
      imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
      imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
      imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
      imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
      imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
    if cmode<0> == '1' && op == '0' then
      imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
      imm64 = Replicate(imm32, 2);
    if cmode<0> == '1' && op == '1' then
      if UsingAArch32() then ReservedEncoding();
      imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:(operand<31:23>), increasedprecision;

    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> :Zeros(48);
  (N-9);

  return imm64; return result;

```

## Library pseudocode for shared/functions/vector/MatMulAddUnsignedSatQ

```
// MatMulAdd()
// =====
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])
// UnsignedSatQ()
// =====

bits(N) (bits(N), boolean) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
    assert N == 128;

    bits(N) result;
    bits(32) sum;
    integer prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = UnsignedSatQ(integer i, integer N)
            integer result;
            boolean saturated;
            if i > 2^N - 1 then
                result = 2^N - 1; saturated = TRUE;
            elsif i < 0 then
                result = 0; saturated = TRUE;
            else
                result = i; saturated = FALSE;
            return (result < N-1:0>, saturated); Elem[addend, 2*i + j, 32];

            for k = 0 to 7
                prod = Int(Elem[op1, 8*i + k, 8], op1_unsigned) * Int(Elem[op2, 8*j + k, 8], op2_unsigned);
                sum = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

## Library pseudocode for shared/functiontrace/vectorCommon/PolynomialMultGetTimestamp

```
// PolynomialMult()
// =====
// GetTimestamp()
// =====
// Returns the Timestamp depending on the type

bits(M+N)bits(64) PolynomialMult(bits(M) op1, bits(N) op2)
    result = GetTimestamp( Timestamp timeStampType)
    case timeStampType of
        when Timestamp_Physical
            return PhysicalCountInt();
        when Timestamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when Timestamp_OffsetPhysical
            return PhysicalCountInt() - CNTPOFF_EL2;
        when Timestamp_None
            return Zeros(M+N);
    extended_op2 = (64);
    when ZeroExtendTimestamp_CoreSight(op2, M+N);
    for i=0 to M-1
        if op1[i] == '1' then
            result = result EOR return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise LSLUnreachable(extended_op2, i);
    return result();
```

## Library pseudocode for shared/functiontrace/vectorselfhosted/SatQEffectiveE0HTRE

```
// SatQ()
// =====
// EffectiveE0HTRE()
// =====
// Returns effective E0HTRE value

(bits(N), boolean)bit SatQ(integer i, integer N, boolean unsigned)
  (result, sat) = if unsigned then EffectiveE0HTRE()
  return if UnsignedSatQELUsingAArch32(i, N) else( SignedSatQEL2(i, N);
  return (result, sat);) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
```

## Library pseudocode for shared/functiontrace/vectorselfhosted/SignedSatQEffectiveE0TRE

```
// SignedSatQ()
// =====
// EffectiveE0TRE()
// =====
// Returns effective E0TRE value

(bits(N), boolean)bit SignedSatQ(integer i, integer N)
  integer result;
  boolean saturated;
  if i > 2^(N-1) - 1 then
    result = 2^(N-1) - 1; saturated = TRUE;
  elsif i < -(2^(N-1)) then
    result = -(2^(N-1)); saturated = TRUE;
  else
    result = i; saturated = FALSE;
  return (result<N-1:0>, saturated); EffectiveE0TRE()
  return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

## Library pseudocode for shared/functiontrace/vectorselfhosted/UnsignedRSqrtEstimateEffectiveE1TRE

```
// UnsignedRSqrtEstimate()
// =====
// EffectiveE1TRE()
// =====
// Returns effective E1TRE value

bits(N)bit UnsignedRSqrtEstimate(bits(N) operand)
  assert N == 32;
  bits(N) result;
  if operand<N-1:N-2> == '00' then // Operands <= 0xFFFFFFFF produce 0xFFFFFFFF
    result = EffectiveE1TRE()
  return if OnesUsingAArch32(N);
  else
    // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
    // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
    increasedprecision = FALSE;
    estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);

  return result;() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

## Library pseudocode for

shared/functiontrace/vectorselfhosted/UnsignedRecipEstimateEffectiveE2TRE

```
// UnsignedRecipEstimate()
// =====
// EffectiveE2TRE()
// =====
// Returns effective E2TRE value

bits(N)bit UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result =EffectiveE2TRE()
    return if OnesUsingAArch32(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0]

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0]
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0]
        result = estimate<8:0> : Zeros(N-9);

    return result;() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

## Library pseudocode for

shared/functiontrace/vectorselfhosted/UnsignedSatQSelfHostedTraceEnabled

```
// UnsignedSatQ()
// =====
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.

(bits(N), boolean)boolean UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1; saturated = TRUE;
    elsif i < 0 then
        result = 0; saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);SelfHostedTraceEnabled()
    if !(HaveTraceExt() && HaveSelfHostedTrace()) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
        if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
    else
        if SecureOnlyImplementation() && !ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;

    return FALSE;
```

## Library pseudocode for shared/trace/Commonselfhosted/GetTimestampTraceAllowed

```
// GetTimestamp()
// TraceAllowed()
// =====
// Returns the Timestamp depending on the type
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

bits(64)boolean GetTimestamp(TraceAllowed(bits(2) el)
    if !TimeStampHaveTraceExt(timestampType)
        case timestampType of
            when() then return FALSE;
            ss = TimeStamp_PhysicalSecurityStateAtEL
                return(el);
            if PhysicalCountIntSelfHostedTraceEnabled();
                when() then
                    boolean trace_allowed;
                    // Detect scenarios where tracing in this Security state is never allowed.
                    case ss of
                        when TimeStamp_VirtualSS_NonSecure
                            return trace_allowed = TRUE;
                        when PhysicalCountIntSS_Secure() - CNTVOFF_EL2;
                            when bit trace_bit;
                                if TimeStamp_OffsetPhysicalHaveEL
                                    return(PhysicalCountIntEL3() - CNTPOFF_EL2;
                                when) then
                                    trace_bit = if TimeStamp_NoneELUsingAArch32
                                        return( ZerosEL3(64);
                                    when) then SDCR.STE else MDCR_EL3.STE;
                                        else
                                            trace_bit = '1';
                                            trace_allowed = trace_bit == '1';
                                bit TRE_bit;
                                case el of
                                    when TimeStamp_CoreSightEL3
                                        return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
                                    otherwise TRE_bit = if !
                                        () then TRFCR.E1TRE else '0';
                                    when EL2 TRE_bit = EffectiveE2TRE();
                                    when EL1 TRE_bit = EffectiveE1TRE();
                                    when EL0
                                        if EffectiveTGE() == '1' then
                                            TRE_bit = EffectiveE0HTRE();
                                        else
                                            TRE_bit = EffectiveE0TRE();
                                return trace_allowed && TRE_bit == '1';
                            else
                                case ss of
                                    when SS_NonSecure return ExternalNoninvasiveDebugEnabled();
                                    when SS_Secure return ExternalSecureNoninvasiveDebugEnabledUnreachableHaveAArch64();
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRETraceContextIDR2

```
// EffectiveE0HTRE()
// =====
// Returns effective E0HTRE value
// TraceContextIDR2()
// =====

bitboolean EffectiveE0HTRE()
    return if TraceContextIDR2()
    if ! ELUsingAArch32TraceAllowed((PSTATE.EL) || !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;() || TRFCR_EL2.CX == '1';
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0TRETraceSynchronizationBarrier

```
// EffectiveE0TRE()
// =====
// Returns effective E0TRE value

bit// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers EffectiveE0TRE()
return ifTraceSynchronizationBarrier(); ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE1TRETraceTimeStamp

```
// EffectiveE1TRE()
// TraceTimeStamp()
// =====
// Returns effective E1TRE value

bitTimeStamp EffectiveE1TRE()
return ifTraceTimeStamp()
if () then
    if HaveEL(EL2) then
        TS_el2 = TRFCR_EL2.TS;
        if !HaveECVExt() && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);

        case TS_el2 of
            when '00'
                // Falls out to check TRFCR_EL1.TS
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;

        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP);

        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt();
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable(); // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSightUsingAArch32SelfHostedTraceEnabled() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

## Library pseudocode for shared/trace/translation/selfhostedat/EffectiveE2TREATAccess

```
// EffectiveE2TRE()
// =====
// Returns effective E2TRE value

bitenumeration EffectiveE2TRE()
return ifATAccess { ATAccess_Read,
    ATAccess_Write,
    ATAccess_ReadPAN,
    UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE; ATAccess_WritePAN
};
```

## Library pseudocode for shared/tracettranslation/selfhostedat/SelfHostedTraceEnabledEncodePARAttrs

```
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.
// EncodePARAttrs()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

booleanbits(8) SelfHostedTraceEnabled()
    if !(EncodePARAttrs(HaveTraceExtMemoryAttributes() && memattrs)
        bits(8) result;

    if HaveSelfHostedTraceHaveMTEExt() then return FALSE;
    if EDSCR.TF0 == '0' then return TRUE;
    if() && memattrs.tagged then
        result<7:0> = '11110000';
        return result;

    if memattrs.memtype == HaveELMemType_Device(then
        result<7:4> = '0000';
        if memattrs.device == EL3DeviceType_nGnRnE) then
            secure_trace_enable = ifthen
                result<3:0> = '0000';
            elsif memattrs.device == ELUsingAArch32DeviceType_nGnRE(then
                result<3:0> = '0100';
            elsif memattrs.device == EL3DeviceType_nGRE) then SDCR.STE else MDCR_EL3.STE;
            if secure_trace_enable == '1' && !then
                result<3:0> = '1000';
            else // DeviceType_GRE
                result<3:0> = '1100';
        else
            if memattrs.outer.attrs == ExternalSecureNoninvasiveDebugEnabledMemAttr_WT() then
                return TRUE;
            else
                ifthen
                    result<7:6> = if memattrs.outer.transient then '00' else '10';
                    result<5:4> = memattrs.outer.hints;
                elsif memattrs.outer.attrs == SecureOnlyImplementationMemAttr_WB() && !then
                    result<7:6> = if memattrs.outer.transient then '01' else '11';
                    result<5:4> = memattrs.outer.hints;
                else // MemAttr_NC
                    result<7:4> = '0100';

            if memattrs.inner.attrs == then
                result<3:2> = if memattrs.inner.transient then '00' else '10';
                result<1:0> = memattrs.inner.hints;
            elsif memattrs.inner.attrs == MemAttr_WBExternalSecureNoninvasiveDebugEnabledMemAttr_WT() then
                return TRUE;
        then
            result<3:2> = if memattrs.inner.transient then '01' else '11';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
            result<3:0> = '0100';

    return FALSE; return result;
```

## Library pseudocode for shared/~~tracetranslation~~/selfhostedat/TraceAllowedPAREncodeShareability

```
// TraceAllowed()
// =====
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.
// PAREncodeShareability()
// =====
// Derive 64-bit PAR SH field.

booleanbits(2) TraceAllowed(bits(2) el)
    if !PAREncodeShareability(HaveTraceExtMemoryAttributes()) then return FALSE;
    ss = memattrs;
    if (memattrs.memtype == SecurityStateAtELMemType_Device(el);
    if ++
        (memattrs.inner.attrs == SelfHostedTraceEnabledMemAttr_NC() then
            boolean trace_allowed;
            // Detect scenarios where tracing in this Security state is never allowed.
            case ss of
                when &&
                    memattrs.outer.attrs == SS_NonSecureMemAttr_NC
                        trace_allowed = TRUE;
                when)) then
                    // Force Outer-Shareable on Device and Normal Non-Cacheable memory
                    return '10';

            case memattrs.shareability of
                when SS_SecureShareability_NSH
                    bit trace_bit;
                    if return '00';
                when HaveELShareability_ISH( return '11';
                when EL3Shareability_OSH) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';

            bit TRE_bit;
            case el of
                when EL3 TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
                when EL2 TRE_bit = EffectiveE2TRE();
                when EL1 TRE_bit = EffectiveE1TRE();
                when EL0
                    if EffectiveTGE() == '1' then
                        TRE_bit = EffectiveE0HTRE();
                    else
                        TRE_bit = EffectiveE0TRE();

            return trace_allowed && TRE_bit == '1';
        else
            case ss of
                when SS_NonSecure return ExternalNoninvasiveDebugEnabled();
                when SS_Secure return ExternalSecureNoninvasiveDebugEnabled(); return '10';
```

## Library pseudocode for shared/~~tracetranslation~~/selfhostedat/TraceContextIDR2TranslationStage

```
// TraceContextIDR2()
// =====

booleanenumeration TraceContextIDR2()
    if !TranslationStage_1(TraceAllowed(PSTATE.EL)) || !TranslationStage_1(HaveEL(EL2)) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1'); TranslationStage_12
};
```

## Library pseudocode for

shared/~~tracettranslation~~/~~selfhostedattr~~/TraceSynchronizationBarrierDecodeDevice

```
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers// DecodeDevice()
// =====
// Decode output Device type

DeviceType
TraceSynchronizationBarrier();DecodeDevice(bits(2) device)
    case device of
        when '00' return DeviceType_nGnRnE;
        when '01' return DeviceType_nGnRE;
        when '10' return DeviceType_nGRE;
        when '11' return DeviceType_GRE;
```

```

// TraceTimeStamp()
// =====
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

TimeStampMemAttrHints TraceTimeStamp()
  if DecodeLDFAttr(bits(4) attr) SelfHostedTraceEnabledMemAttrHints() then
    if ldfattr;

    if attr IN {'x0xx'} then ldfattr.attrs = HaveELMemAttr_WT(); // Write-through
    elsif attr == '0100' then ldfattr.attrs = EL2MemAttr_NC() then
      TS_el2 = TRFCR_EL2.TS;
      if !; // Non-cacheable
    elsif attr IN {'x1xx'} then ldfattr.attrs = HaveECVExtMemAttr_WB() && TS_el2 == '10' then
      // Reserved value
      (-, TS_el2) =; // Write-back
    else ConstrainUnpredictableBitsUnreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != Unpredictable_EL2TIMESTAMPMemAttr_NC);

    case TS_el2 of
      when '00'
        // Falls out to check TRFCR_EL1.TS
      when '01'
        return then
    case attr<1:0> of
      when '00' ldfattr.hints = TimeStamp_VirtualMemHint_No;
      when '10'
        assert; // No allocation hints
      when '01' ldfattr.hints = HaveECVExtMemHint_WA(); // Otherwise ConstrainUnpredictableBits re
        return; // Write-allocate
      when '10' ldfattr.hints = TimeStamp_OffsetPhysicalMemHint_RA;
      when '11'
        return; // Read-allocate
      when '11' ldfattr.hints = TimeStamp_PhysicalMemHint_RWA;
    ; // Read/Write allocate

    TS_el1 = TRFCR_EL1.TS;
    if TS_el1 == '00' || (! // The Transient hint applies only to cacheable memory with some alloc
    if ldfattr.attrs != HaveECVExtMemAttr_NC() && TS_el1 == '10') then
      // Reserved value
      (-, TS_el1) = && ldfattr.hints != ConstrainUnpredictableBitsMemHint_No(Unpredictable_EL1TIMEST

    case TS_el1 of
      when '01'
        return TimeStamp_Virtual;
      when '10'
        assert HaveECVExt();
        return TimeStamp_OffsetPhysical;
      when '11'
        return TimeStamp_Physical;
      otherwise
        Unreachable(); // ConstrainUnpredictableBits removes this case
    else
      return TimeStamp_CoreSight; then
      ldfattr.transient = attr<3> == '0';

    return ldfattr;

```

## Library pseudocode for shared/translation/atattrs/ATAccessDecodeSDFAttr

```
enumeration// DecodeSDFAttr()  
// =====  
// Decode memory attributes using SDF (Short Descriptor Format) mapping  
  
MemAttrHints ATAccess {DecodeSDFAttr(bits(2) rgn)  
    ATAccess_Read,sdfattr;  
  
    case rgn of  
        when '00' // Non-cacheable (no allocate)  
            sdfattr.attrs =  
                ATAccess_Write,;  
        when '01' // Write-back, Read and Write allocate  
            sdfattr.attrs =  
                ATAccess_ReadPAN,;  
            sdfattr.hints =  
                ;  
        when '10' // Write-through, Read allocate  
            sdfattr.attrs = MemAttr_WT;  
            sdfattr.hints = MemHint_RA;  
        when '11' // Write-back, Read allocate  
            sdfattr.attrs = MemAttr_WB;  
            sdfattr.hints = MemHint_RAATAccess_WritePAN  
    };  
  
    sdfattr.transient = FALSE;  
  
    return sdfattr;
```

## Library pseudocode for shared/translation/atattrs/EncodePARAttrDecodeShareability

```
// EncodePARAttr()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.
// DecodeShareability()
// =====
// Decode shareability of target memory region

bits(8) Shareability EncodePARAttr(DecodeShareability(bits(2) sh)
  case sh of
    when '10' return MemoryAttributesShareability_OSH memattrs)
  bits(8) result;

  if;
    when '11' return HaveMTEExtShareability_ISH() && memattrs.tagged then
      result<7:0> = '11110000';
      return result;

  if memattrs.memtype ==;
    when '00' return MemType_DeviceShareability_NSH then
      result<7:4> = '0000';
      if memattrs.device ==;
        otherwise
          case DeviceType_nGnRnEConstrainUnpredictable then
            result<3:0> = '0000';
          elsif memattrs.device == ( DeviceType_nGnREUnpredictable_Shareability then
            result<3:0> = '0100';
          elsif memattrs.device ==) of
            when DeviceType_nGREConstraint_OSH then
              result<3:0> = '1000';
            else // DeviceType_GRE
              result<3:0> = '1100';

  else
    if memattrs.outer.attrs == return MemAttr_WTShareability_OSH then
      result<7:6> = if memattrs.outer.transient then '00' else '10';
      result<5:4> = memattrs.outer.hints;
    elsif memattrs.outer.attrs ==;
      when MemAttr_WBConstraint_ISH then
        result<7:6> = if memattrs.outer.transient then '01' else '11';
        result<5:4> = memattrs.outer.hints;
      else // MemAttr_NC
        result<7:4> = '0100';

    if memattrs.inner.attrs == return MemAttr_WTShareability_ISH then
      result<3:2> = if memattrs.inner.transient then '00' else '10';
      result<1:0> = memattrs.inner.hints;
    elsif memattrs.inner.attrs ==;
      when return Shareability_NSHMemAttr_WBConstraint_NSH then
        result<3:2> = if memattrs.inner.transient then '01' else '11';
        result<1:0> = memattrs.inner.hints;
      else // MemAttr_NC
        result<3:0> = '0100';

  return result;;
```

## Library pseudocode for shared/translation/atattrs/PAREncodeShareabilityEffectiveShareability

```
// PAREncodeShareability()
// EffectiveShareability()
// =====
// Derive 64-bit PAR SH field.
// Force Outer Shareability on Device and Normal iNCoNC memory

bits(2) Shareability PAREncodeShareability(EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
         memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when _____ return Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';
    else
        return memattrs.shareability;
```

## Library pseudocode for shared/translation/atattrs/TranslationStageMAIRAttr

```
enumeration// MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) TranslationStage {MAIRAttr(integer index,
    TranslationStage_1,
    TranslationStage_12
};mair)
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;
```

## Library pseudocode for shared/translation/attrs/DecodeDeviceNormalNCMemAttr

```
// DecodeDevice()
// =====
// Decode output Device type
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes

DeviceTypeMemoryAttributes DecodeDevice(bits(2) device)
    case device of
        when '00' returnNormalNCMemAttr() DeviceType_nGnRnEMemAttrHints;
        when '01' returnnon_cacheable;
        non_cacheable.attrs = DeviceType_nGnREMemAttr_NC;
        when '10' return; DeviceType_nGREMemoryAttributes;
        when '11' returnnc_memattrs;
        nc_memattrs.memtype = ;
        nc_memattrs.outer = non_cacheable;
        nc_memattrs.inner = non_cacheable;
        nc_memattrs.shareability = Shareability_OSHDeviceType_GREMemType_Normal;;
        nc_memattrs.tagged = FALSE;

    return nc_memattrs;
```

## Library pseudocode for shared/translation/ attrs/DecodeLDFAttrS1ConstrainUnpredictableRESMAIR

```
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

MemAttrHints boolean DecodeLDFAttr(bits(4) attr) S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean slaarch64)
    case attr of
        when '0000xx01' return !(slaarch64 && MemAttrHintsHaveFeatXS) ldfattr;

    if attr IN {'x0xx'} then ldfattr.attrs = ();
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(slaarch64 && MemAttr_WTHaveFeatXS); // Write-through
    elsif attr == '0100' then ldfattr.attrs = ();
        when '10100000' return !(slaarch64 && MemAttr_NCHaveFeatXS); // Non-cacheable
    elsif attr IN {'x1xx'} then ldfattr.attrs = ();
        when '11110000' return !(slaarch64 && MemAttr_WBHaveMTE2Ext); // Write-back
    else
        Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No; // No allocation hints
            when '01' ldfattr.hints = MemHint_WA; // Write-allocate
            when '10' ldfattr.hints = MemHint_RA; // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;();
    when 'xxxx0000' return TRUE;
    otherwise return FALSE;
```



```

// DecodeSDFAttr()
// =====
// Decode memory attributes using SDF (Short Descriptor Format) mapping
// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

MemAttrHintsMemoryAttributes DecodeSDFAttr(bits(2) rgn)S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, bool
bits(8) attr = attr_in;
if
MemAttrHintsS1ConstrainUnpredictableRESMAIR sdfattr;

case rgn of
when '00' // Non-cacheable (no allocate)
sdfattr.attrs = {attr, slaarch64} then
(-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

MemoryAttributes memattrs;
case attr of
when '0000xxxx' // Device memory
memattrs.memtype = MemType_Device;
memattrs.device = DecodeDevice(attr<3:2>);
memattrs.tagged = FALSE;
memattrs.xs = if slaarch64 then NOT attr<0> else '1';
when '01000000'
assert slaarch64 && HaveFeatXS();
memattrs.memtype = MemType_Normal;
memattrs.tagged = FALSE;
memattrs.outer.attrs = MemAttr_NC;
when '01' // Write-back, Read and Write allocate
sdfattr.attrs = memattrs.inner.attrs = MemAttr_WBMemAttr_NC;
sdfattr.hints = memattrs.xs = '0';

when '10100000'
assert slaarch64 && MemHint_RWHaveFeatXS;
when '10' // Write-through, Read allocate
sdfattr.attrs = ();
memattrs.memtype = MemType_Normal;
memattrs.tagged = FALSE;
memattrs.outer.attrs = MemAttr_WT;
sdfattr.hints = memattrs.outer.hints = MemHint_RA;
when '11' // Write-back, Read allocate
sdfattr.attrs = memattrs.outer.transient = FALSE;
memattrs.inner.attrs = MemAttr_WT;
memattrs.inner.hints = MemHint_RA;
memattrs.inner.transient = FALSE;
memattrs.xs = '0';
when '11110000' // Tagged memory
assert slaarch64 && HaveMTE2Ext();
memattrs.memtype = MemType_Normal;
memattrs.tagged = TRUE;
memattrs.outer.attrs = MemAttr_WB;
sdfattr.hints = memattrs.outer.hints = ;
memattrs.outer.transient = FALSE;
memattrs.inner.attrs = MemAttr_WB;
memattrs.inner.hints = MemHint_RWA;
memattrs.inner.transient = FALSE;
memattrs.xs = '0';
otherwise
memattrs.memtype = MemType_Normal;
memattrs.outer = DecodeLDFAttr(attr<7:4>);
memattrs.inner = DecodeLDFAttr(attr<3:0>);
memattrs.tagged = FALSE;

if (memattrs.inner.attrs == MemAttr_WB &&
memattrs.outer.attrs == MemAttr_WB) then
memattrs.xs = '0';
else
memattrs.xs = '1';

```

```

memattrs.shareability = DecodeShareabilityMemHint_RAMemHint_RWA;
(sh);

sdfattr.transient = FALSE;

return sdfattr; return memattrs;

```

### Library pseudocode for shared/translation/attrs/DecodeShareabilityS2CombineS1AttrHints

```

// DecodeShareability()
// =====
// Decode shareability of target memory region
// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

ShareabilityMemAttrHints DecodeShareability(bits(2) sh)
  case sh of
    when '10' return S2CombineS1AttrHints( Shareability_OSHMemAttrHints;
    when '11' returns1_attrhints, Shareability_ISHMemAttrHints;
    when '00' returns2_attrhints) Shareability_NSHMemAttrHints;
    otherwise
      case attrhints;

    if s1_attrhints.attrs == ConstrainUnpredictableMemAttr_NC(|| s2_attrhints.attrs == Unpredictable_Share
      when then
        attrhints.attrs = Constraint_OSHMemAttr_NC return;
    elsif s1_attrhints.attrs == Shareability_OSHMemAttr_WT;
      when || s2_attrhints.attrs == Constraint_ISHMemAttr_WT return then
        attrhints.attrs = Shareability_ISHMemAttr_WT;
      when else
        attrhints.attrs = Constraint_NSHMemAttr_WB return;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != Shareability_NSHMemAttr_NC; then
      attrhints.hints = s1_attrhints.hints;
      attrhints.transient = s1_attrhints.transient;

    return attrhints;

```

### Library pseudocode for shared/translation/attrs/EffectiveShareabilityS2CombineS1Device

```

// EffectiveShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory
// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

ShareabilityDeviceType EffectiveShareability(S2CombineS1Device(MemoryAttributesDeviceType memattrs)
  if (memattrs.memtype == s1_device, MemType_DeviceDeviceType ||
    (memattrs.inner.attrs == s2_device)
  if s1_device == MemAttr_NCDeviceType_nGnRnE &&
    memattrs.outer.attrs == || s2_device == MemAttr_NCDeviceType_nGnRnE)) then
  then
    return ;
  elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
    return DeviceType_nGnRE;
  elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
    return DeviceType_nGRE;
  else
    return DeviceType_GREShareability_OSHDeviceType_nGnRnE;
  else
    return memattrs.shareability;

```

## Library pseudocode for shared/translation/attrs/MAIRAttrS2CombineS1MemAttrs

```
// MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR
// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

bits(8) MemoryAttributes MAIRAttr(integer index, S2CombineS1MemAttrs( s1_memattrs,
                                                                    MemoryAttributes s2_memattrs)
    MemoryAttributes memattrs;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    if ELUsingAArch32(EL2) || !HaveMTE2Ext() then
        memattrs.tagged = FALSE;
    else
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);
    memattrs.xs = s2_memattrs.xs;

    memattrs.shareability = EffectiveShareabilityMAIRTypeMemoryAttributes mair)
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;(memattrs);
    return memattrs;
```

## Library pseudocode for shared/translation/attrs/NormalNCMemAttrS2CombineS1Shareability

```
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

MemoryAttributesShareability NormalNCMemAttr()S2CombineS1Shareability(
    MemAttrHintsShareability non_cacheable;
    non_cacheable.attrs = s1_shareability, MemAttr_NCShareability; s2_shareability)

    if (s1_shareability ==
        MemoryAttributesShareability_OSH nc_memattrs;
        nc_memattrs.memtype = ||
            s2_shareability == MemType_NormalShareability_OSH;
        nc_memattrs.outer = non_cacheable;
        nc_memattrs.inner = non_cacheable;
        nc_memattrs.shareability = ) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
    nc_memattrs.tagged = FALSE;

    return nc_memattrs;;
```

## Library pseudocode for shared/translation/ attrs/S1ConstrainUnpredictableRESMAIRS2DecodeCacheability

```
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

boolean MemAttrHints S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean slaarch64)
    case attr of
        when '0000xx01' return !(slaarch64 && S2DecodeCacheability(bits(2) attr) HaveFeatXSMemAttrHints())
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(slaarch64 && s2attr;

    case attr of
        when '01' s2attr.attrs = HaveFeatXSMemAttr_NC();
        when '10100000' return !(slaarch64 && ; // Non-cacheable
        when '10' s2attr.attrs = HaveFeatXSMemAttr_WT();
        when '11110000' return !(slaarch64 && ; // Write-through
        when '11' s2attr.attrs = ; // Write-back
        otherwise // Constrained unpredictable
            case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
                when Constraint_NC s2attr.attrs = MemAttr_NC;
                when Constraint_WT s2attr.attrs = MemAttr_WT;
                when Constraint_WB s2attr.attrs = MemAttr_WBHaveMTE2ExtMemAttr_WB();
        when 'xxxx0000' return TRUE;
        otherwise return FALSE;;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```



```

// S1DecodeMemAttrs()
// S2DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1
// Decode stage 2 memory attributes

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean slaarch64)
    bits(8) attr = attr_in;
    if S2DecodeMemAttrs(bits(4) attr, bits(2) sh) S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    MemoryAttributes memattrs;

    case attr of
        when '0000xxxx' // Device memory
            memattrs.memtype = when '00xx' // Device memory
                memattrs.memtype = MemType_Device;
            memattrs.device = DecodeDevice(attr<3:2>);
            memattrs.tagged = FALSE;
            memattrs.xs = if slaarch64 then NOT attr<0> else '1';
        when '01000000'
            assert slaarch64 &&(attr<1:0>);
        otherwise // Normal memory
            memattrs.memtype = HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged = FALSE;
            memattrs.outer.attrs = memattrs.outer = MemAttr_NCS2DecodeCacheability;
            memattrs.inner.attrs = (attr<3:2>);
            memattrs.inner = MemAttr_NCS2DecodeCacheability;
            memattrs.xs = '0';

        when '10100000'
            assert slaarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged = FALSE;
            memattrs.outer.attrs = MemAttr_WT;
            memattrs.outer.hints = MemHint_RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs = MemAttr_WT;
            memattrs.inner.hints = MemHint_RA;
            memattrs.inner.transient = FALSE;
            memattrs.xs = '0';
        when '11110000' // Tagged memory
            assert slaarch64 && HaveMTE2Ext();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged = TRUE;
            memattrs.outer.attrs = MemAttr_WB;
            memattrs.outer.hints = MemHint_RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs = MemAttr_WB;
            memattrs.inner.hints = MemHint_RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs = '0';
        otherwise
            memattrs.memtype = MemType_Normal;
            memattrs.outer = DecodeLDFAttr(attr<7:4>);
            memattrs.inner = DecodeLDFAttr(attr<3:0>);
            memattrs.tagged = FALSE;

            if (memattrs.inner.attrs == MemAttr_WB &&
                memattrs.outer.attrs == MemAttr_WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';
    (attr<1:0>);

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;

```

## Library pseudocode for shared/translation/attrs/S2CombineS1AttrHintsWalkMemAttrs

```
// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.
// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk

MemAttrHintsMemoryAttributes S2CombineS1AttrHints(WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn) MemAttrHintsMemoryAttributes s1_attrhints, MemAttrHintsMemoryAttributes s2_attrhints)
{
    walkmemattrs.memtype = MemAttrHintsMemType_Normal s2_attrhints;
    walkmemattrs.shareability = MemAttrHintsDecodeShareability attrhints;

    if s1_attrhints.attrs == (sh);
    walkmemattrs.inner = MemAttr_NCDecodeSDFAttr || s2_attrhints.attrs == (irgn);
    walkmemattrs.outer = MemAttr_NCDecodeSDFAttr then
        attrhints.attrs = (orgn);
    walkmemattrs.tagged = FALSE;
    if (walkmemattrs.inner.attrs == MemAttr_NCMemAttr_WB;
    elsif s1_attrhints.attrs == &&
        walkmemattrs.outer.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;
    ) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';

    return attrhints; return walkmemattrs;
}
```

## Library pseudocode for shared/translation/attrsfaults/S2CombineS1DeviceAlignmentFault

```
// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2
// AlignmentFault()
// =====

DeviceTypeFaultRecord S2CombineS1Device(AlignmentFault(DeviceTypeAccType s1_device, acctype, boolean iswrite) DeviceTypeFaultRecord s1_fault, AlignmentFault(DeviceTypeAccType s2_device, acctype, boolean iswrite) DeviceTypeFaultRecord s2_fault)
{
    if s1_device == fault;
        fault.statuscode = DeviceType_nGnRnEFault_Alignment || s2_device == DeviceType_nGnRnE then
            return DeviceType_nGnRnE;
        elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
            return DeviceType_nGnRE;
        elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
            return DeviceType_nGRE;
        else
            return DeviceType_GRE;
        fault.acctype = acctype;
        fault.write = iswrite;
        fault.secondstage = secondstage;

    return fault;
}
```

```

// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes
// AsyncExternalAbort()
// =====
// Return a fault record indicating an asynchronous external abort

MemoryAttributesFaultRecord S2CombineS1MemAttrs(AsyncExternalAbort(boolean parity, bits(2) errortype, bit(1)
    fault.statuscode = if parity then
        MemoryAttributesFault_AsyncParity s2_memattrs) else
        MemoryAttributesFault_AsyncExternal memattrs;

    if s1_memattrs.memtype ==;
        fault.extflag = extflag;
        fault.errortype = errortype;
        fault.acctype = MemType_DeviceAccType_NORMAL && s2_memattrs.memtype == MemType_Device then
            memattrs.memtype = MemType_Device;
            memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
        elsif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
            memattrs = s1_memattrs;
        elsif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
            memattrs = s2_memattrs;
        else // S2 Normal, S1 Normal
            memattrs.memtype = MemType_Normal;
            memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
            memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    if ELUsingAArch32(EL2) || !HaveMTE2Ext() then
        memattrs.tagged = FALSE;
    else
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);
    memattrs.xs = s2_memattrs.xs;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;

    return fault;

```

## Library pseudocode for shared/translation/attrsfaults/S2CombineS1ShareabilityNoFault

```
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

ShareabilityFaultRecord S2CombineS1Shareability(NoFault()ShareabilityFaultRecord s1_shareability, fault;

    fault.statuscode =

        ShareabilityFault_None s2_shareability)

    if (s1_shareability ==;
    fault.acctype = Shareability_OSHAccType_NORMAL ||
        s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
        s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    return fault;
```

## Library pseudocode for shared/ translation/attrtranslation/S2DecodeCacheabilityS1TranslationRegime

```
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

MemAttrHints bits(2) S2DecodeCacheability(bits(2) attr) S1TranslationRegime(bits(2) el)
    if el !=
        MemAttrHints EL0 s2attr;

    case attr of
        when '01' s2attr.attrs = then
            return el;
        elsif MemAttr_NCHaveEL; // Non-cacheable
            when '10' s2attr.attrs = ( MemAttr_WTEL3; // Write-through
            when '11' s2attr.attrs = ) && MemAttr_WBELUsingAArch32; // Write-back
            otherwise // Constrained unpredictable
                case ( ConstrainUnpredictableEL3() && SCR.NS == '0' then
                    return Unpredictable_S2RESMEMATTREL3) of
                        when;
        elsif Constrain_NCHaveVirtHostExt s2attr.attrs = ( ) && MemAttr_NCELIsInHost;
            when (el) then
                return Constraint_WTEL2 s2attr.attrs =;
            else
                return MemAttr_WTEL1;
            when
// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) Constraint_WB s2attr.attrs = S1TranslationRegime()
    return MemAttr_WBS1TranslationRegime;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr; (PSTATE.EL);
```

## Library pseudocode for shared/translation/attrs/vmsa/S2DecodeMemAttrsAddressDescriptor

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes

MemoryAttributes type S2DecodeMemAttrs(bits(4) attr, bits(2) sh) AddressDescriptor is (
    FaultRecord fault, // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype = memattrs, MemType_DeviceFullAddress;
            memattrs.device = paddress,
            bits(64) vaddress
    )

constant integer DecodeDevice(attr<1:0>);
    otherwise // Normal memory
        memattrs.memtype = MemType_Normal;
        memattrs.outer = S2DecodeCacheability(attr<3:2>);
        memattrs.inner = S2DecodeCacheability(attr<1:0>);

    memattrs.shareability = DecodeShareability(sh);

    return memattrs; FINAL_LEVEL = 3;
```

## Library pseudocode for shared/translation/attrs/vmsa/WalkMemAttrsContiguousSize

```
// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk
// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range

MemoryAttributes integer WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn) ContiguousSize(
    MemoryAttributes TGx walkmemattrs;

    walkmemattrs.memtype = tgx, integer level)
    case tgx of
        when MemType_Normal TGx_4KB;
            walkmemattrs.shareability = assert level IN {1, 2, 3};
            return 4;
        when DecodeShareability TGx_16KB(sh);
            walkmemattrs.inner = assert level IN {2, 3};
            return if level == 2 then 5 else 7;
        when DecodeSDFAttr TGx_64KB(irgn);
            walkmemattrs.outer = DecodeSDFAttr(orgn);
            walkmemattrs.tagged = FALSE;
            if (walkmemattrs.inner.attrs == MemAttr_WB &&
                walkmemattrs.outer.attrs == MemAttr_WB) then
                walkmemattrs.xs = '0';
            else
                walkmemattrs.xs = '1';

    return walkmemattrs; assert level IN {2, 3};
    return 5;
```

## Library pseudocode for shared/translation/faults/vmsa/AlignmentFaultCreateAddressDescriptor

```
// AlignmentFault()
// =====
// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

FaultRecordAddressDescriptor AlignmentFault(CreateAddressDescriptor(bits(64) va, AccTypeFullAddress acctype,
    FaultRecordMemoryAttributes fault;

    fault.statuscode = memattrs) addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault = NoFaultFault_AlignmentAddressDescriptor;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = secondstage;
());

return fault; return addrdesc;
```

## Library pseudocode for shared/translation/faults/vmsa/AsyncExternalAbortCreateFaultyAddressDescriptor

```
// AsyncExternalAbort()
// =====
// Return a fault record indicating an asynchronous External abort
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

FaultRecordAddressDescriptor AsyncExternalAbort(boolean parity, bits(2) errortype, bit extflag)CreateFaultyAddressDescriptor(
    FaultRecord fault;

    fault.statuscode = if parity then fault) Fault_AsyncParityAddressDescriptor else Fault_AsyncExternal;
    fault.extflag = extflag;
    fault.errortype = errortype;
    fault.acctype = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
addrdesc;

    return fault; addrdesc.vaddress = va;
    addrdesc.fault = fault;

    return addrdesc;
```

## Library pseudocode for shared/translation/faults/vmsa/NoFaultDescriptorType

```
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

FaultRecordenumeration NoFault()DescriptorType {
    FaultRecord fault;

    fault.statuscode = DescriptorType_Table, Fault_None;
    fault.acctype = DescriptorType_Block, DescriptorType_Page,
    AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    return fault;DescriptorType_Invalid
};
```

## Library pseudocode for shared/translation/translationvmsa/S1TranslationRegimeDomains

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

bits(2) constant bits(2) S1TranslationRegime(bits(2) el)
    if el != Domain_NoAccess == '00';
constant bits(2) EL0 then
    return el;
    elsif Domain_Client == '01';
constant bits(2) HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
    return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
    return EL2;
    else
    return EL1;

// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL); Domain_Manager == '11';
```

## Library pseudocode for shared/translation/vmsa/AddressDescriptorFetchDescriptor

```
type// FetchDescriptor()
// =====
// Fetch a translation table descriptor

(FaultRecord, bits(N)) AddressDescriptor is (FetchDescriptor(bit ee,
    AddressDescriptor walkaddress, FaultRecord fault, // fault.statuscode indicates whether the
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    assert N == 32 || N == 64;
    bits(N) descriptor;
    MemoryAttributes FaultRecord memattrs, fault = fault_in;
    FullAddressAccessDescriptor paddress,
    bits(64) vaddress
)
walkacc;

constant integer walkacc.acctype = ;
// MPAM PARTID for translation table walk is determined by the access invoking the translation
walkacc.mpam = GenMPAMcurEL(fault.acctype);

PhysMemRetStatus memstatus;
(memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
if IsFault(memstatus) then
    fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress,
    walkacc, N DIV 8, fault);
    if IsFault(fault.statuscode) then
        return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(FINAL_LEVEL = 3; (descriptor));

    return (fault, descriptor);
```

## Library pseudocode for shared/translation/vmsa/ContiguousSizeHasUnprivileged

```
// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

integer boolean ContiguousSize(HasUnprivileged(TGxRegime_tgx, integer level)
    case tgx of
        when regime)
            return (regime IN { TGx_4KBRegime_EL20
                assert level IN {1, 2, 3};
                return 4;
            when, TGx_16KBRegime_EL30
                assert level IN {2, 3};
                return if level == 2 then 5 else 7;
            when, TGx_64KBRegime_EL10
                assert level IN {2, 3};
                return 5;});
```

## Library pseudocode for shared/translation/vmsa/CreateAddressDescriptorIsAtomicRW

```
// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values
// IsAtomicRW()
// =====
// Is the access an atomic operation?

AddressDescriptor boolean CreateAddressDescriptor(bits(64) va, IsAtomicRW( FullAddressAccType pa, acctype)
    return acctype IN {
        MemoryAttributesAccType_ATOMICRW memattrs),
        AddressDescriptorAccType_ORDEREDRW addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault = NoFaultAccType_ORDEREDATOMICRW();

    return addrdesc;};
```

## Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptorRegime

```
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

AddressDescriptor enumeration CreateFaultyAddressDescriptor(bits(64) va, Regime { FaultRecord fault) Regime_
    Regime_EL30, // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2, // EL2
    Regime_EL20, // EL2&0
    AddressDescriptor addrdesc;

    addrdesc.vaddress = va;
    addrdesc.fault = fault;

    return addrdesc; Regime_EL10 // EL1&0
};
```

## Library pseudocode for shared/translation/vmsa/DescriptorTypeRegimeUsingAArch32

```

enumeration// RegimeUsingAArch32()
// =====
// Determine if the EL controlling the regime executes in AArch32 state

boolean DescriptorType {RegimeUsingAArch32(
    DescriptorType_Table, regime)
    case regime of
        when
            DescriptorType_Block, return
            DescriptorType_Page, {
                };
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2 return ELUsingAArch32(EL2);
        when Regime_EL3DescriptorType_Invalid
    }; return FALSE;

```

## Library pseudocode for shared/translation/vmsa/DomainsS1TTWParams

```

constant bits(2) type Domain_NoAccess = '00';
constant bits(2) S1TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,          // TCR_ELx.HA
    bit      hd,          // TCR_ELx.HD
    bit      tbi,         // TCR_ELx.TBI{x}
    bit      tbiid,       // TCR_ELx.TBID{x}
    bit      e0pd,        // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit      ds,          // TCR_ELx.DS
    bits(3)  ps,          // TCR_ELx.{I}PS
    bits(6)  txsz,        // TCR_ELx.TxSZ
    bit      epan,        // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit      dct,         // HCR_EL2.DCT
    bit      nv1,         // HCR_EL2.NV1
    bit      cmow,        // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'

// A32-VMSA exclusive parameters
    bits(3)  t0sz,        // TTBCR.T0SZ
    bits(3)  t1sz,        // TTBCR.T1SZ
    bit      uwxn,        // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32) Domain_Client = '01';
constant bits(2) t0gx,    // TCR_ELx.TGx / Always TGx_4KB
    bits(2)  irgn,        // TCR_ELx.IRGNx / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)  orgn,        // TCR_ELx.ORGNx / TTBCR.ORGnx or HTCR.ORGn0
    bits(2)  sh,          // TCR_ELx.SHx / TTBCR.SHx or HTCR.SH0
    bit      hpd,         // TCR_ELx.HPD{x} / TTBCR2.HPDx or HTCR.HPD
    bit      ee,          // SCTLR_ELx.EE / SCTLR.EE or HSCTLR.EE
    bit      wxn,         // SCTLR_ELx.WXN / SCTLR.WXN or HSCTLR.WXN
    bit      ntlsmd,      // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit      dc,          // HCR_EL2.DC / HCR.DC
    bit      sif,         // SCR_EL3.SIF / SCR.SIF Domain_Manager = '11'; mair // MAIR_ELx
)

```

```

// FetchDescriptor()
// =====
// Fetch a translation table descriptor

(FaultRecord, bits(N))type FetchDescriptor(bit ee, S2TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,      // VTCR_EL2.HA
    bit      hd,      // VTCR_EL2.HD
    bit      sl2,     // V{S}TCR_EL2.SL2
    bit      ds,      // VTCR_EL2.DS
    bit      sw,      // VSTCR_EL2.SW
    bit      nsw,     // VTCR_EL2.NSW
    bit      sa,      // VSTCR_EL2.SA
    bit      nsa,     // VTCR_EL2.NSA
    bits(3)   ps,      // VTCR_EL2.PS
    bits(6)   txsz,    // V{S}TCR_EL2.T0SZ
    bit      fw,      // HCR_EL2.PTW
    bit      cmow,    // HCRX_EL2.CMOW

// A32-VMSA exclusive parameters
    bit      s,       // VTCR.S
    bits(4)   t0sz,   // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32) AddressDescriptorTGx walkaddress
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    assert N == 32 || N == 64;
    bits(N) descriptor;
    FaultRecord fault = fault_in;
    AccessDescriptor walkacc;

    walkacc.acctype = AccType_TTW;
    // MPAM PARTID for translation table walk is determined by the access invoking the translation
    walkacc.mpam = GenMPAMcurEL(fault.acctype);

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
    if IsFault(memstatus) then
        fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress,
                                     walkacc, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor); t0, // V{S}TCR_EL2.T0 / Always TGx_4KB
    bits(2)   sl0,    // V{S}TCR_EL2.SL0 / VTCR.SL0
    bits(2)   irgn,   // VTCR_EL2.IRGN0 / VTCR.IRGN0
    bits(2)   orgn,   // VTCR_EL2.ORGNO / VTCR.ORGNO
    bits(2)   sh,     // VTCR_EL2.SH0 / VTCR.SH0
    bit      ee,     // SCTL_EL2.EE / HSCTLR.EE
    bit      ptw,    // HCR_EL2.PTW / HCR.PTW
    bit      vm,     // HCR_EL2.VM / HCR.VM
)

```

## Library pseudocode for shared/translation/vmsa/HasUnprivilegedSDFTYPE

```
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean enumeration HasUnprivileged(SDFTYPE {Regime regime)
    return (regime IN {SDFTYPE_Table,
        Regime_EL20, SDFTYPE_Invalid,
        Regime_EL30, SDFTYPE_Supersection,
        SDFTYPE_Section,
        SDFTYPE_LargePage,
        Regime_EL10
    }); SDFTYPE_SmallPage
};
```

## Library pseudocode for shared/translation/vmsa/IsAtomicRWSecurityStateForRegime

```
// IsAtomicRW()
// =====
// Is the access an atomic operation?
// SecurityStateForRegime()
// =====
// Return the Security State of the given translation regime

boolean SecurityState IsAtomicRW(SecurityStateForRegime(AccTypeRegime acctype)
    return acctype IN {regime}
    case regime of
        when
            AccType_ATOMICRWRegime_EL3, return
            AccType_ORDEREDRWSecurityStateAtEL, {
                };
            when Regime_EL30 return SS_Secure; // A32 EL3 is always Secure
            when Regime_EL2 return SecurityStateAtEL(EL2);
            when Regime_EL20 return SecurityStateAtEL(EL2);
            when Regime_EL10 return SecurityStateAtEL(EL1AccType_ORDEREDATOMICRWEL3
                });};
```

## Library pseudocode for shared/translation/vmsa/RegimeStageOA

```
enumeration // StageOA()
// =====
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress Regime {StageOA(bits(64) ia,
    Regime_EL3, // EL3tgx,
    Regime_EL30, // EL3&0 (PL1&0 when EL3 is AArch32)walkstate)
    // Output Address
    Regime_EL2, // EL2oa;
    integer csize;

    tsize =
    Regime_EL20, // EL2&0(tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize =
        Regime_EL10 // EL1&0
    };(tgx, walkstate.level);
    else
        csize = 0;

    ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<51:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

## Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32TGx

```
// RegimeUsingAArch32()
// =====
// Determine if the EL controlling the regime executes in AArch32 state

boolean enumeration RegimeUsingAArch32(TGx {Regime regime})
    case regime of
        when TGx_4KB, Regime_EL10 return TGx_16KB, ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2 return ELUsingAArch32(EL2);
        when Regime_EL3 return FALSE; TGx_64KB
};
```

## Library pseudocode for shared/translation/vmsa/S1TTWParamsTGxGranuleBits

```
type// TGxGranuleBits()
// =====
// Retrieve the address size, in bits, of a granule

integer S1TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,          // TCR_ELx.HA
    bit      hd,          // TCR_ELx.HD
    bit      tbi,         // TCR_ELx.TBI{x}
    bit      tbid,        // TCR_ELx.TBID{x}
    bit      e0pd,        // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit      ds,          // TCR_ELx.DS
    bits(3)   ps,          // TCR_ELx.{I}PS
    bits(6)   txsz,        // TCR_ELx.TxSZ
    bit      epan,        // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit      dct,         // HCR_EL2.DCT
    bit      nv1,         // HCR_EL2.NV1
    bit      cmow,        // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'

// A32-VMSA exclusive parameters
    bits(3)   t0sz,        // TTBCR.T0SZ
    bits(3)   t1sz,        // TTBCR.T1SZ
    bit      uwxn,        // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32) TGxGranuleBits(
    TGx      tgx,          // TCR_ELx.TGx / Always TGx_4KB
    bits(2)   irgn,        // TCR_ELx.IRGNx / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)   orgn,        // TCR_ELx.ORGNg / TTBCR.ORGNg or HTCR.ORGNO
    bits(2)   sh,          // TCR_ELx.SHx / TTBCR.SHx or HTCR.SHO
    bit      hpd,          // TCR_ELx.HPD{x} / TTBCR2.HPDx or HTCR.HPD
    bit      ee,           // SCTLR_ELx.EE / SCTLR.EE or HSCTLR.EE
    bit      wxn,          // SCTLR_ELx.WXN / SCTLR.WXN or HSCTLR.WXN
    bit      ntlsmd,       // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit      dc,           // HCR_EL2.DC / HCR.DC
    bit      sif,          // SCR_EL3.SIF / SCR.SIFtgx)
    case tgx of
        when
            return 12;
        when TGx_16KB return 14;
        when TGx_64KBMAIRTypeTGx_4KB mair // MAIR_ELx / MAIR1:MAIR0 or HMAIR1:HMAIR0
    )return 16;
```

## Library pseudocode for shared/translation/vmsa/S2TTWParamsTLBContext

```

type S2TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,      // VTCR_EL2.HA
    bit      hd,      // VTCR_EL2.HD
    bit      sl2,     // V{S}TCR_EL2.SL2
    bit      ds,      // VTCR_EL2.DS
    bit      sw,      // VSTCR_EL2.SW
    bit      nsw,     // VTCR_EL2.NSW
    bit      sa,      // VSTCR_EL2.SA
    bit      nsa,     // VTCR_EL2.NSA
    bits(3)  ps,      // VTCR_EL2.PS
    bits(6)  txsz,    // V{S}TCR_EL2.T0SZ
    bit      fwb,     // HCR_EL2.PTW
    bit      cmow,    // HCRX_EL2.CMOW

// A32-VMSA exclusive parameters
    bit      s,       // VTCR.S
    bits(4)  t0sz,    // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    SecurityState ss,
    Regime      regime,
    bits(16)    vmid,
    bits(16)    asid,
    bit         nG,
    PAspace     ipaspace, // Used in stage 2 lookups & invalidations only
    boolean     includes_s1,
    boolean     includes_s2,
    bits(64)    ia,      // Input Address
    TGx         tgx,     // V{S}TCR_EL2.TG0 / Always TGx_4KB
    bits(2)     sl0,     // V{S}TCR_EL2.SL0 / VTCR.SL0
    bits(2)     irgn,    // VTCR_EL2.IRGN0 / VTCR.IRGN0
    bits(2)     orgn,    // VTCR_EL2.ORGNO / VTCR.ORGNO
    bits(2)     sh,      // VTCR_EL2.SH0 / VTCR.SH0
    bit         ee,      // SCTLR_EL2.EE / HSCTLR.EE
    bit         ptw,     // HCR_EL2.PTW / HCR.PTW
    bit         vm       // HCR_EL2.VM / HCR.VM

    tg,
    bit         cnp,
    bit         xs       // XS attribute (FEAT_XS)
)

```

## Library pseudocode for shared/translation/vmsa/SDFTYPETLBRecord

```

enumerationtype SDFTYPE {TLBRecord is (
    SDFTYPE_Table, context,
    SDFTYPE_Invalid,
    SDFTYPE_Supersection,
    SDFTYPE_Section,
    SDFTYPE_LargePage,
    SDFTYPE_SmallPage
};walkstate,
    integer     blocksize, // Number of bits directly mapped from IA to OA
    integer     contigsize, // Number of entries log 2 marking a contiguous output range
    bits(64)    s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(64)    s2descriptor // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)

```

## Library pseudocode for shared/translation/vmsa/SecurityStateForRegimeTTWState

```
// SecurityStateForRegime()
// =====
// Return the Security State of the given translation regime

SecurityStateType SecurityStateForRegime(TTWState is (
    boolean          istable,
    integer          level, RegimeFullAddress regime)
    case regime of
        when baseaddress,
            bit          contiguous,
            bit          nG,
            bit          guardedpage, Regime_EL3SDFType returnsdftype, // AArch32 Short-descriptor for
            bits(4)      domain, // AArch32 Short-descriptor format walk only SecurityStateAtELMemoryAtt
            when Regime_EL30 return SS_Secure; // A32 EL3 is always Secure
            when Regime_EL2 return SecurityStateAtEL(EL2);
            when Regime_EL20 return SecurityStateAtEL(EL2);
            when Regime_EL10 return SecurityStateAtEL(EL1);permissions
)
)
```

## Library pseudocode for shared/translation/vmsa/StageOATranslationRegime

```
// StageOA()
// =====
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address
// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

FullAddressRegime StageOA(bits(64) ia, TranslationRegime(bits(2) el, TGxAccType tgx, acctype)
    if el == TTWStateEL3 walkstate)
    // Output Address then
    return if
        FullAddressELUsingAArch32 oa;
    integer csize;

    tsize = ( TranslationSizeEL3(tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize = ) then else Regime_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    elsif el == EL1 then
        if acctype == AccType_NV2REGISTER then
            assert EL2Enabled();
            return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
        else
            return Regime_EL10;
    elsif el == EL0 then
        if IsSecure() && ELUsingAArch32(EL3) then
            return Regime_EL30;
        elsif ELIsInHost(EL0) then
            return Regime_EL20;
        else
            return Regime_EL10;
    else
        UnreachableContiguousSizeRegime_EL30(tgx, walkstate.level);
    else
        csize = 0;

    ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<51:ia_msb>;ia<ia_msb-1:0>;

    return oa;());
```

## Library pseudocode for shared/translation/vmsa/TGxTranslationSize

```

enumeration// TranslationSize()
// =====
// Compute the number of bits directly mapped from the input address
// to the output address

integer TGx {TranslationSize(
    TGx_4KB,tgx, integer level)
    granulebits =
    TGx_16KB,(tgx);
    blockbits = (
    TGx_64KB
    );
    level) * (granulebits - 3);
};

return granulebits + blockbits;

```

## Library pseudocode for shared/translation/vmsa/TGxGranuleBitsUseASID

```

// TGxGranuleBits()
// =====
// Retrieve the address size, in bits, of a granule
// UseASID()
// =====
// Determine whether the translation context for the access requires ASID or is a global entry

integerboolean TGxGranuleBits(UseASID(TGxTLBContext tgx)
    case tgx of
        whenaccess)
    return TGx_4KBHasUnprivileged return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;(access.regime);

```

## Library pseudocode for shared/translation/vmsa/TLBContextUseVMID

```

type// UseVMID()
// =====
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean TLBContext is (UseVMID(
    SecurityStateTLBContext ss,access)
    return access.regime ==
    RegimeRegime_EL10 regime,
    bits(16) vmid,
    bits(16) asid,
    bit nG,&&
    PASpaceEL2Enabled ipaspace, // Used in stage 2 lookups & invalidations only
    boolean includes_s1,
    boolean includes_s2,
    bits(64) ia, // Input Address
    TGx tg,
    bit cnp,
    bit xs // XS attribute (FEAT_XS)
)());

```

## Library pseudocode for shared/translation/vmsa/TLBRecordVARange

```

typeenumeration TLBRecord is (VARange {
    TLBContext context,VARange_LOWER,
    TTWState walkstate,
    integer blocksize, // Number of bits directly mapped from IA to OA
    integer contigsize, // Number of entries log 2 marking a contiguous output range
    bits(64) s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(64) s2descriptor // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)VARange_UPPER
};

```

## Library pseudocode for shared/translation/vmsa/TTWState

```
type TTWState is (  
    boolean          istable,  
    integer          level,  
    FullAddress      baseaddress,  
    bit              contiguous,  
    bit              nG,  
    bit              guardedpage,  
    SDFType          sdftype, // AArch32 Short-descriptor format walk only  
    bits(4)          domain, // AArch32 Short-descriptor format walk only  
    MemoryAttributes memattrs,  
    Permissions      permissions  
)
```

## Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()  
// =====  
// Select the translation regime given the target EL and PE state  
  
Regime TranslationRegime(bits(2) el_in, AccType acctype)  
  
    bits(2) el;  
    case acctype of  
        when AccType_NV2REGISTER  
            assert EL2Enabled() && el_in == EL1;  
            el = EL2;  
        otherwise  
            el = el_in;  
  
    if el == EL3 then  
        return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;  
    elsif el == EL2 then  
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;  
    elsif el == EL1 then  
        return Regime_EL10;  
    elsif el == EL0 then  
        if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3) then  
            return Regime_EL30;  
        elsif ELIsInHost(EL0) then  
            return Regime_EL20;  
        else  
            return Regime_EL10;  
    else  
        Unreachable();
```

## Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()  
// =====  
// Compute the number of bits directly mapped from the input address  
// to the output address  
  
integer TranslationSize(TGx tgx, integer level)  
    granulebits = TGxGranuleBits(tgx);  
    blockbits   = (FINAL_LEVEL - level) * (granulebits - 3);  
  
    return granulebits + blockbits;
```

## Library pseudocode for shared/translation/vmsa/UseASID

```
// UseASID()  
// =====  
// Determine whether the translation context for the access requires ASID or is a global entry  
  
boolean UseASID(TLBContext access)  
    return HasUnprivileged(access.regime);
```

Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()  
// =====  
// Determine whether the translation context for the access requires VMID to match a TLB entry  
  
boolean UseVMID(TLBContext access)  
    return access.regime == Regime_EL10 && EL2Enabled();
```

Library pseudocode for shared/translation/vmsa/VARange

```
enumeration VARange {  
    VARange_LOWER,  
    VARange_UPPER  
};
```

Internal version only: isa v01\_27v01\_26, pseudocode v2022-03\_relv2021-12 to suppress diffs in 2022-03-RC1; Build timestamp: 2022-03-29T10:20:4611

Copyright © 2010-20222010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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