

ARM® Architecture Reference Manual

Supplement

ARMv8.1, for ARMv8-A architecture profile

This document is now RETIRED.
The Arm Architecture Reference Manual for A-profile architecture (DDI0487) is the definitive reference for this architecture specification.



ARM Architecture Reference Manual Supplement

ARMv8.1, for ARMv8-A architecture profile

Copyright © 2016 ARM Limited or its affiliates. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
29 April 2016	A.a	Non-Confidential	EAC release, limited circulation
03 June 2016	A.b	Non-Confidential	EAC release

Proprietary Notice

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

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

This document may include technical inaccuracies or typographical errors.

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 shall prevail.

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

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

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

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

Copyright © 2016 ARM Limited or its affiliates. All rights reserved.
ARM Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

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

Confidentiality Status

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

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

RETIRED

RETIRED

Contents

ARM Architecture Reference Manual Supplement ARMv8.1, for ARMv8-A architecture profile

Preface

About this supplement	x
Using this book	xi
Conventions	xiv
Additional reading	xv
Feedback	xvi

Part A ARMv8.1 Architecture Introduction and Overview

Chapter A1

Introduction

A1.1	About this ARMv8.1 supplement	A1-20
A1.2	About the ARMv8.1 architecture	A1-21

Part B ARMv8.1 Changes in the AArch64 Architecture

Chapter B1

New Atomic Instructions

B1.1	About atomic instructions	B1-28
------	---------------------------------	-------

Chapter B2

AArch64 SIMD Instructions for Rounding Double Multiply Add/Subtract

B2.1	About the new instructions	B2-32
------	----------------------------------	-------

Chapter B3

Hierarchical Permission Disables

B3.1	About Hierarchical Permission Disables	B3-34
------	--	-------

Chapter B4	Hardware Updates to Access Flag and Dirty State	
B4.1	About hardware management of the Access flag and of dirty state	B4-36
Chapter B5	AArch64 Privileged Access Never	
B5.1	About the Privileged Access Never (PAN) bit	B5-50
Chapter B6	Limited Ordering Regions	
B6.1	About limited ordering regions	B6-54
Chapter B7	16-bit VMID	
B7.1	VMID size	B7-58
Chapter B8	Virtualization Host Extensions	
B8.1	About the Virtualization Host Extensions	B8-60
Chapter B9	AArch64 Performance Monitors Extension	
B9.1	Changes to the Performance Monitors Extension	B9-74
Chapter B10	A64 Instruction Set Encoding	
B10.1	Loads and stores	B10-78
B10.2	Data processing - SIMD and floating point	B10-81
Chapter B11	A64 Instructions	
B11.1	Alphabetical list of instructions	B11-86
B11.2	ARMv8.0 sections relating to these instructions	B11-227
Chapter B12	AArch64 Register Descriptions	
B12.1	General information about AArch64 System registers	B12-232
B12.2	General system control registers	B12-234
B12.3	Debug registers	B12-424
B12.4	Performance Monitors registers	B12-450
B12.5	Generic Timer registers	B12-463
B12.6	System instructions	B12-504
B12.7	ARMv8.0 sections relating to these registers	B12-545

Part C **ARMv8.1 Changes in the AArch32 Architecture**

Chapter C1	A32/T32 Advanced SIMD Instructions for Rounding Double Multiply Add/Subtract	
C1.1	About the new instructions	C1-562
Chapter C2	AArch32 Privileged Access Never	
C2.1	About the Privileged Access Never bit	C2-564
Chapter C3	AArch32 Performance Monitors Extension	
C3.1	Changes to the Performance Monitors Extension	C3-568
Chapter C4	A32/T32 Instruction Set Encoding	
C4.1	Advanced SIMD data-processing	C4-572
Chapter C5	A32 and T32 Instructions	
C5.1	Alphabetical list of instructions	C5-576
C5.2	ARMv8.0 sections relating to these instructions	C5-584

Chapter C6	AArch32 Register Descriptions	
C6.1	General information about AArch32 System registers	C6-590
C6.2	General system control registers	C6-591
C6.3	Debug registers	C6-644
C6.4	Performance Monitors registers	C6-673
C6.5	Generic Timer registers	C6-687
C6.6	ARMv8.0 sections relating to these registers	C6-698

Part D **ARMv8.1 Changes to External Debug**

Chapter D1	PC Sample-based Profiling	
D1.1	Changes to PC Sample-based profiling	D1-704
Chapter D2	External Debug Register Descriptions	
D2.1	General information about External debug register descriptions	D2-706
D2.2	Debug registers	D2-707
D2.3	Performance Monitors registers	D2-733

Part E **Architectural Pseudocode**

Chapter E1	ARMv8.1 Pseudocode	
E1.1	About the ARMv8.1 pseudocode chapter	E1-746
E1.2	Library pseudocode for AArch64	E1-747
E1.3	Library pseudocode for AArch32	E1-807
E1.4	Common library pseudocode	E1-875

Part F **Appendixes**

Appendix A	Notes on Using Debug and Performance Monitors	
F1.1	Self-hosted debug	F1-952
F1.2	External debug	F1-953
F1.3	Performance Monitors	F1-954

RETIRED

Preface

This preface introduces the *ARM Architecture Reference Manual Supplement, ARMv8.1, for ARMv8-A architecture profile*. It contains the following sections:

- *About this supplement* on page x.
- *Using this book* on page xi.
- *Conventions* on page xiv.
- *Additional reading* on page xv.
- *Feedback* on page xvi.

About this supplement

This supplement describes the changes that are introduced by ARM® architecture v8.1, ARMv8.1. For a summary of these changes, see [Chapter A1 Introduction](#).

This supplement must be read with the latest version of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. For more information about the information included in this supplement, and how this supplement relates to the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, see [About this ARMv8.1 supplement on page A1-20](#).

Some sections from the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* have been included in this supplement to complement instruction and register descriptions.

This manual is organized into parts as described in [Using this book on page xi](#).

RETIRED

Using this book

The purpose of this book is to describe the changes that are introduced in ARMv8.1. It is a supplement to the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* (DDI0487), version A.j, and is intended to be used with it. There might be inconsistency between this supplement and the ARMv8 Architecture Reference Manual due to some late-breaking changes. Therefore, the ARMv8-A ARM is the definitive source of information about ARMv8.0.

It is assumed that the reader is familiar with the ARMv8 architecture.

The information in this book is organized into parts, as described in this section.

Part A, Introduction and Architecture Overview

Part A contains the following chapters:

Chapter A1 Introduction

Read this for a summary of how this supplement relates to the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* and an introduction to the ARMv8.1 architecture.

Part B, ARMv8.1 Changes in the AArch64 Architecture

Part B describes the changes to the architecture in AArch64 state. It contains the following chapters:

Chapter B1 New Atomic Instructions

Read this for a description of the new atomic instructions added to the A64 instruction set.

Chapter B2 AArch64 SIMD Instructions for Rounding Double Multiply Add/Subtract

Read this for a description of the Rounding double multiply-add and Rounding double multiply-subtract instructions added to the A64 Advanced SIMD instruction set.

Chapter B3 Hierarchical Permission Disables

Read this for a description of the Hierarchical permission disables feature of ARMv8.1 in AArch64 state.

Chapter B4 Hardware Updates to Access Flag and Dirty State

Read this for a description of the hardware management of the Access flag and dirty state feature of ARMv8.1 in AArch64 state.

Chapter B5 AArch64 Privileged Access Never

Read this for a description of the addition of a Privileged Access Never field to PSTATE.

Chapter B6 Limited Ordering Regions

Read this for a description of the Limited ordering regions (LORegions) feature of ARMv8.1.

Chapter B7 16-bit VMID

Read this for a description of the 16-bit VMID feature of ARMv8.1.

Chapter B8 Virtualization Host Extensions

Read this for a description the Virtualization Host Extensions feature of ARMv8.1, that add enhanced support for type 2 hypervisors to operation in AArch64 state.

Chapter B9 AArch64 Performance Monitors Extension

Read this for a description of the changes to the Performance Monitors Extension introduced in ARMv8.1.

Chapter B10 A64 Instruction Set Encoding

Read this for a description of the encoding of the instructions that ARMv8.1 adds to the A64 instruction set.

Chapter B11 A64 Instructions

Read this for descriptions of the new A64 instructions that are introduced in ARMv8.1.

Chapter B12 AArch64 Register Descriptions

Read this for descriptions of the AArch64 System registers that are added or affected by ARMv8.1.

Part C, ARMv8.1 Changes in the AArch32 Architecture

Part C describes the changes to the architecture in AArch32 state. It contains the following chapters:

Chapter C1 A32/T32 Advanced SIMD Instructions for Rounding Double Multiply Add/Subtract

Read this for a description of the Rounding double multiply-add and Rounding double multiply-subtract instructions added to the T32 and A32 Advanced SIMD instruction sets.

Chapter C2 AArch32 Privileged Access Never

Read this for a description of the addition of a Privileged Access Never field to PSTATE.

Chapter C3 AArch32 Performance Monitors Extension

Read this for a description of the changes to the Performance Monitors Extension introduced in ARMv8.1.

Chapter C4 A32/T32 Instruction Set Encoding

Read this for a description of the encoding of the instructions that ARMv8.1 adds to the T32 and A32 instruction sets.

Chapter C5 A32 and T32 Instructions

Read this for descriptions of the new T32 and A32 instructions that are introduced in ARMv8.1.

Chapter C6 AArch32 Register Descriptions

Read this for descriptions of the AArch32 System registers that are added or affected by ARMv8.1.

Part D, ARMv8.1 Changes to External Debug

Part D describes the external debug registers. It contains the following chapters:

Chapter D1 PC Sample-based Profiling

Read this for a description of the changes to the PC Sample-based Profiling Extension in ARMv8.1.

Chapter D2 External Debug Register Descriptions

Read this for descriptions of the External debug registers that are added or affected by ARMv8.1, or by the changes to the Performance Monitors Extension introduced with ARMv8.1.

Part E, Architectural Pseudocode

Part E contains pseudocode that describes various features of the ARM architecture. It contains the following chapter:

Chapter E1 ARMv8.1 Pseudocode

Read this for the definition of pseudocode that describes various features of the ARMv8.1 architecture, for operation in AArch64 and in AArch32 state, including a summary of the changes that are made by the introduction of ARMv8.1.

Part F, Appendixes

This manual contains the following appendix:

Appendix F1 *Notes on Using Debug and Performance Monitors*

Read this for a description of aspects of the use of Debug, including the Performance Monitors, that are affected by ARMv8.1 or by the changes to the Performance Monitors introduced with ARMv8.1.

RETIRED

Conventions

The following sections describe conventions that this book can use:

- [Typographic conventions](#).
- [Numbers](#).
- [Pseudocode descriptions](#).
- [Assembler syntax descriptions](#).

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

bold Denotes signal names, and is used for terms in descriptive lists, where appropriate.

monospace Used for assembler syntax descriptions, pseudocode, and source code examples.
Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

Colored text Indicates a link. This can be:

- A URL, for example <http://infocenter.arm.com>.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, [ARM publications on page xv](#).
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example [Chapter B4](#).

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by 0x. In both cases, the prefix and the associated value are written in a monospace font, for example 0xFFFF0000. To improve readability, long numbers can be written with an underscore separator between every four characters, for example 0xFFFF_0000_0000_0000. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font, and is described in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a monospace font, and use the conventions described in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

Additional reading

This section lists relevant publications from ARM and third parties.

See the Infocenter <http://infocenter.arm.com>, for access to ARM documentation.

ARM publications

- *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487).

RETIRED

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title.
- The number, DDI0557A.b.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

———— **Note** ————

ARM tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

RETIRED

Part A

ARMv8.1 Architecture Introduction and Overview

RETIRED

RETIRED

Chapter A1

Introduction

This chapter summarizes how this supplement relates to the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* and introduces the ARMv8.1 architecture. It contains the following sections:

- [About this ARMv8.1 supplement on page A1-20.](#)
- [About the ARMv8.1 architecture on page A1-21.](#)

A1.1 About this ARMv8.1 supplement

This supplement must be read with the most recent issue of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. Together, that manual and this supplement provide a full description of the ARMv8-A architecture, as extended by the ARMv8.1 architecture extensions.

[About the ARMv8.1 architecture on page A1-21](#) introduces the ARMv8.1 architecture extensions.

In general, this supplement describes only the architectural changes that are introduced by the ARMv8.1 architecture extensions. The major exceptions to this approach are:

- Except for ESR_ELx, this supplement includes the complete description of each register and System instruction that is changed by ARMv8.1. See:
 - [Chapter B12 AArch64 Register Descriptions](#).
 - [Chapter C6 AArch32 Register Descriptions](#).
 - [Chapter D2 External Debug Register Descriptions](#).

———— **Note** ————

[The register descriptions included in this supplement on page B12-232](#) indicates why ESR_ELx is the exception to the general approach.

- To complement the register descriptions, and the descriptions of the new instructions introduced by ARMv8.1, some sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* are included in this supplement. These sections have been updated to include any changes made by ARMv8.1, see:
 - [ARMv8.0 sections relating to these instructions on page B11-227](#).
 - [ARMv8.0 sections relating to these registers on page B12-545](#).
 - [ARMv8.0 sections relating to these registers on page C6-698](#).
- This supplement includes the complete ARMv8 pseudocode library, as updated for ARMv8.1. See [Chapter E1 ARMv8.1 Pseudocode](#).

A1.2 About the ARMv8.1 architecture

ARMv8.1 introduces a set of extensions to the ARMv8-A architecture profile. This document includes the definitions of features. The new features that are supported are described under the following headings:

- [Features supported by AArch64](#).
- [Features supported by AArch32 on page A1-22](#).

A1.2.1 Features supported by AArch64

Updates to the ISA

- A set of atomic read/write instructions. See [Chapter B1 New Atomic Instructions](#).
- Two new SIMD instructions, SQRDMLAH, and SQRDMLSH. See [Chapter B2 AArch64 SIMD Instructions for Rounding Double Multiply Add/Subtract](#).
- A set of load and store instructions that provide memory order over a localized address range of memory. See [Chapter B6 Limited Ordering Regions](#).
- Changes to the MRS and MSR instructions to support the PSTATE.PAN state bit. See [Chapter B5 AArch64 Privileged Access Never](#).
- All implementations of the ARMv8.1 architecture are required to implement the CRC32 instructions. These are OPTIONAL in ARMv8.0.

Changes to the ARMv8 memory model

- A facility to disable the hierarchical attributes, APTable, PXNTable, UXNTable in the translation table is added. See [Chapter B3 Hierarchical Permission Disables](#).
- Support for hardware management of the Access flag and Dirty bit in the AArch64 translation table is introduced. See [Chapter B4 Hardware Updates to Access Flag and Dirty State](#).
- A new PAN (Privileged Access Never) state bit is added to PSTATE. A corresponding bit is added to the SPSRs. See [Chapter B5 AArch64 Privileged Access Never](#).
- *Limited ordering regions* (LORegions) allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the physical address map as observed by a limited set of observers. See [Chapter B6 Limited Ordering Regions](#).

Changes to virtualization

- The option of using a 16-bit VMID. See [Chapter B7 16-bit VMID](#).
- Virtualization Host Extensions introduced for implementations with EL2, to provide support for Type 2 hypervisors in Non-secure state. See [Chapter B8 Virtualization Host Extensions](#).

Enhancements to the Performance Monitor Extensions

Enhancements to the Performance Monitors Extension. In an ARMv8.1 implementation, the STALL_FRONTEND and STALL_BACKEND events must be implemented. The event space is extended to 16 bits. See [Chapter B9 AArch64 Performance Monitors Extension](#).

The following table shows the fields that identify when an ARMv8.1 feature in the AArch64 Execution state is enabled.

Feature	Identification mechanism	Value
New atomic instructions	ID_AA64ISAR0_EL1.Atomic	0010
SIMD instructions for rounding double multiply operations	ID_AA64ISAR0_EL1.RDM	0001
Hierarchical permission disables	ID_AA64MMFR1_EL1.HD	0001

Feature	Identification mechanism	Value
Hardware updates to Access flag and dirty state bit	ID_AA64MMFR1_EL1.HAFDBS	0001 - Access flag only. 0010 - Access flag and DBM bit.
Privileged Access Never	ID_AA64MMFR1_EL1.PAN	0001
LORegions	ID_AA64MMFR1_EL1.LO	0001
16-bit VMID	ID_AA64MMFR1_EL1.VMIDBits	0000 - 8 bits 0001 - 16 bits
Virtualization Host Extensions	ID_AA64MMFR1_EL1.VH	0001
	ID_AA64DFR0_EL1.DebugVer	0111
	ID_DFR0_EL1.CopDbg	0111
Performance Monitors Extension	ID_AA64DFR0_EL1.PMUVer	0100

A1.2.2 Features supported by AArch32

The following features are supported by the AArch32 Execution state:

- Two new SIMD instructions, VQRDLAH, and VQRDLASH. See [Chapter C1 A32/T32 Advanced SIMD Instructions for Rounding Double Multiply Add/Subtract](#).
- A new *Privileged Access Never* (PAN) bit introduced in PSTATE, and the associated changes to the ARMv8-A memory model. A corresponding bit is added to the SPSRs and CPSR. A new instruction SETPAN added. See [Chapter C2 AArch32 Privileged Access Never](#).
- Enhancements to the Performance Monitors Extension. In an ARMv8.1 implementation, the STALL_FRONTEND and STALL_BACKEND events must be implemented. The event space is extended to 16 bits. See [Chapter C3 AArch32 Performance Monitors Extension](#).
- Virtualization Host Extensions have an impact of some AArch32 debug registers.

The following table shows the fields that identify when an ARMv8.1 feature in the AArch32 Execution state is enabled.

Feature	Identification mechanism	Value
SIMD instructions for rounding double multiply operations	ID_ISAR5.RDM	0001
Privileged Access Never	ID_MMFR3.PAN	0001
Performance Monitors Extension	ID_DFR0.PerfMon	0100

A1.2.3 Updates to External debug register descriptions

This document also includes the full descriptions of the registers that are accessible through the external debug interface that are affected by the introduction of ARMv8.1, or by the changes to the Performance Monitors Extension introduced with ARMv8. See [Chapter D2 External Debug Register Descriptions](#).

A1.2.4 Dependencies

ETMv4.1 or later is required where trace is part of an ARMv8.1 implementation that supports EL2:

- Virtualization Host Extensions.
- The optional 16-bit VMID.

A1.2.5 See also

In the ARM Architecture Reference Manual

- Introduction to the ARMv8 Architecture.

RETIRED

RETIRED

Part B

ARMv8.1 Changes in the AArch64 Architecture

RETIRED

Chapter B1

New Atomic Instructions

This chapter describes the new atomic instructions added to the A64 instruction set. It contains the following section:

- [About atomic instructions on page B1-28.](#)

B1.1 About atomic instructions

ARMv8.1 introduces a set of atomic instructions.

The atomic instructions introduced are:

- Compare and Swap instructions, CAS, and CASP. These instructions read one or two values from memory, compare those values with other values supplied by the instruction, and if the comparison succeeds, writes back different values, also supplied by the instruction. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write.
- Atomic memory operation instructions, LD<OP>, and ST<OP>, where <OP> is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, and UMIN. Each instruction atomically loads a value from memory, performs an operation on the values, and stores the result back to memory. The LD<OP> instructions leave the originally read value in the destination register of the instruction.
- Swap instruction, SWP. This instruction atomically reads a location from memory into a register and writes back a different supplied value back to the same memory location.

These instructions must be aligned to the total size of the memory location being accessed. Otherwise, an Alignment fault is generated.

For the purpose of permission checking, and for watchpoints, all the new atomic instructions are treated as performing both a load and a store.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the ARMv8-A architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity.

For the CAS and CASP instructions, the architecture permits that a data read clears any exclusive monitors associated with that location, even if the compare subsequently fails. If these instructions generate a synchronous Data Abort, the registers which are compared and loaded are restored to the values held in the registers before the instruction was executed.

The ST<OP> instructions are not regarded as doing a read for the purpose of a DMB LD barrier.

The new instructions are only added to the A64 instruction set. See [Loads and stores on page B10-78](#) for information on the instruction group and instruction classes that the new instructions belong to.

B1.1.1 Behavior in Debug state

In Debug state, these instructions execute as in Non-debug state.

B1.1.2 Impact on Performance Monitors

The Performance Monitors Extension defines events that count load and store instructions, and events that count memory-read and memory-write operations.

For the purpose of Performance Monitors, the LD<OP>, CAS, and SWP instructions are both load and store instructions, and generate both read and write operations. These instructions generate:

- Both LD_RETIRE and ST_RETIRE events.
- The L1D_CACHE_RD event if the memory-read operation accesses the Level 1 data cache.
- The L1D_CACHE_WR event if the memory-write operation accesses the Level 1 data cache.

For the purpose of Performance Monitors, the ST<OP> instructions are store instructions. They generate write operations. It is IMPLEMENTATION DEFINED whether the instruction generates a read operation at any given point in the memory hierarchy as the atomic operation might be performed beyond that point. These instructions generate:

- The ST_RETIRE event.
- The L1D_CACHE_WR event if the operation accesses the Level 1 data cache.

The ST<OP> instructions do not generate the LD_RETIRE event. It is IMPLEMENTATION DEFINED whether the L1D_CACHE_RD event is generated.

For the following events that count cache refills, if an atomic instruction generates one cache refill, only one event is generated:

- L1I_CACHE_REFILL.
- L1D_CACHE_REFILL.
- L2I_CACHE_REFILL.
- L2D_CACHE_REFILL.
- L3D_CACHE_REFILL.

B1.1.3 ESR_ELx fault codes for atomic instructions

In the event of a Data Abort from an atomic instruction:

- The ESR_ELx.WnR field is set to 0 if a read of the location would have generated a fault, otherwise it is set to 1. This field is UNKNOWN for an external abort.
- The ESR_ELx.ISV field is set to 0.

In the event of a Watchpoint from an atomic instruction, the ESR_ELx.WnR field is set to 0 if a read of the location would have generated a fault, otherwise it is set to 1.

B1.1.4 Possible implementation restrictions on using atomic instructions

In some implementations, and for some memory types, the properties of atomicity can be met only by functionality outside the PE. Some system implementations might not support atomic instructions for all regions of the memory. In particular, this can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Device, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such implementations, it is defined by the system:

- Whether the atomic instructions are atomic in regard to other agents that access memory.
- If the atomic instructions are atomic in regard to other agents that access memory, which address ranges or memory types this applies to.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the atomic instructions will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

If the atomic instructions are not atomic in regard to other agents that access memory, then performing an atomic instruction to such a location can have one or more of the following effects:

- The instruction generates a Synchronous external abort.
- The instruction generates a System Error interrupt.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the new Fault Status code of ESR_ELx.DFSC = 110101 for Data Aborts.

For the Non-secure EL1&0 translation regime, if the atomic instruction is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.

- The instruction is treated as a NOP.

- The instructions are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a System Error interrupt.

B1.1.5 Identification mechanism

The [ID_AA64ISAR0_EL1](#).Atomic field identifies the presence of the atomic instructions.

B1.1.6 See also

In this supplement

- [ID_AA64ISAR0_EL1](#).Atomic.
- The following instructions in [Chapter B11 A64 Instructions](#):
 - CAS and CASP instructions.
 - LD<OP> instructions.
 - ST<OP> instructions.
 - SWP instructions.
- [Load/store exclusive on page B10-78](#).
- [Atomic memory operations on page B10-80](#).

In the ARM Architecture Reference Manual

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Loads and stores.
- Load/store exclusive.

Chapter B2

AArch64 SIMD Instructions for Rounding Double Multiply Add/Subtract

This chapter describes the Rounding double multiply add and Rounding double multiply subtract instructions added to the A64 Advanced SIMD instruction set. It contains the following section:

- [About the new instructions on page B2-32.](#)

B2.1 About the new instructions

The following instructions are added to the AArch64 SIMD instruction set:

- SQRDMLAH, Signed Saturating Rounding Doubling Multiply Accumulate Returning High Half.
- SQRDMLSH, Signed Saturating Rounding Doubling Multiply Subtract Returning High Half.

These new instructions are added to the Data processing - SIMD and floating-point instruction group. See [Data processing - SIMD and floating point on page B10-81](#) for information on the instruction classes these new instructions belong to.

B2.1.1 Behavior in Debug state

In Debug state, these instructions are CONSTRAINED UNPREDICTABLE, and the behavior is one of the following:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction executes as in Non-debug state.

B2.1.2 Identification mechanism

The [ID_AA64ISAR0_EL1](#).RDM field identifies the presence of the Rounding Double Multiply Add/Subtract instructions.

B2.1.3 See also

In this supplement

- [Data processing - SIMD and floating point on page B10-81](#).
- [SQRDMLAH \(by element\)](#).
- [SQRDMLAH \(vector\)](#).
- [SQRDMLSH \(by element\)](#).
- [SQRDMLSH \(vector\)](#).
- [ID_AA64ISAR0_EL1](#).RDM.

In the ARM Architecture Reference Manual

The following sections in the A64 Instruction Set Overview chapter of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Data processing - SIMD and floating-point.
- Advanced SIMD scalar three same.
- Advanced SIMD scalar x indexed element.
- Advanced SIMD three same.
- Advanced SIMD vector x indexed element.

Chapter B3

Hierarchical Permission Disables

This chapter describes the Hierarchical permission disables feature of ARMv8.1. It contains the following section:

- [About Hierarchical Permission Disables on page B3-34.](#)

B3.1 About Hierarchical Permission Disables

The hierarchical attributes in the translation tables, APTable, PXNTable, and UXNTable permit subtrees of the translation tables to be used by different agents. Not all operating systems use this functionality, and so ARMv8.1 adds a facility to disable these bits.

When these bits are disabled:

- The value is IGNORED by hardware, allowing them to be used by software.
- The behavior of the system is as if the bits are all set to 0.

This ability to disable hierarchical attribute bits has no effect on the NSTable bit.

Where hierarchical attributes are disabled for the EL2 translation regime, when HCR_EL2.{E2H,TGE} is not {1,1}, and the EL3 translation regime, bit[61] and bit[59] of the next level descriptor attributes in the translation tables are required to be IGNORED by hardware and are no longer reserved, meaning these bits can be used by software.

The Hierarchical permission disables feature is added only to the AArch64 translation regimes.

B3.1.1 Identification mechanism

The ID_AA64MMFR1_EL1.HADS field identifies the support for Hierarchical Permission Disables.

B3.1.2 See also

In this supplement

- ID_AA64MMFR1_EL1.HD.
- TCR_EL1.{HPD1, HPD0}.
- TCR_EL2.HPD.
- TCR_EL3.HPD.

In the ARM Architecture Reference Manual

The following sections in The AArch64 Virtual Memory System Architecture chapter of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Translation tables and the translation process.
- Memory attribute fields in the VMSAv8-64 translation table format descriptors.

Chapter B4

Hardware Updates to Access Flag and Dirty State

This chapter describes the hardware management of Access flag and dirty state feature of ARMv8.1. It contains the following section:

- [About hardware management of the Access flag and of dirty state on page B4-36.](#)

B4.1 About hardware management of the Access flag and of dirty state

In ARMv8.0, all updates to the translation tables are performed by software. ARMv8.1 introduces the following optional features that perform hardware updates to the translation tables, in AArch64 state only:

Hardware management of the Access flag

When enabled, this feature means that, in situations where, without this feature, an Access flag fault would be generated, the hardware instead performs an atomic read-modify-write of the appropriate translation table descriptor, to update the Access flag from 0 to 1.

See [Hardware management of the Access flag on page B4-39](#) for the architectural specification of the feature.

Hardware management of dirty state

In order to support the hardware management of dirty state, a new field, the DBM field, is added to the translation table descriptors as part of ARMv8.1 architecture.

When enabled, the hardware management of dirty state means that, if the Block or Page descriptor in a translation table indicates that a data access does not have write permission, then in situations where, without this feature, a data access would generate a Permission fault only because of this lack of write permission, the hardware checks the value of the DBM field in the Block or Page descriptor. If this field is 1, then instead of generating a Permission fault the hardware performs an atomic read-modify-write of the translation table descriptor, to change the value of the bit that prohibits the write access.

Hardware management of dirty state can only be enabled when hardware management of the Access flag is also enabled.

See [Hardware management of dirty state on page B4-40](#) for the architectural specification of this feature.

Configuration fields are added to:

- [TCR_EL1](#), [TCR_EL2](#), [TCR_EL3](#), and [VTCR_EL2](#) to enable these features.
- [ID_AA64MMFR1_EL1](#) to indicate the level of support for these features.

B4.1.1 Changes to the translation tables

To support hardware management of dirty state, ARMv8.1 introduces the DBM bit in the translation table Block and Page descriptors. When hardware management of dirty state is enabled, this bit indicates whether hardware should perform the associated update to the permission bit in the descriptor. In each of these descriptors, a new bit, bit[51] is allocated as the DBM bit. This bit is RES0 in ARMv8.0.

Figure B4-1 shows the ARMv8.1 level 0, level 1, and level 2 descriptor formats. Bits[63:51] of the descriptor define the memory attributes, and bits[50:48] are RES0.

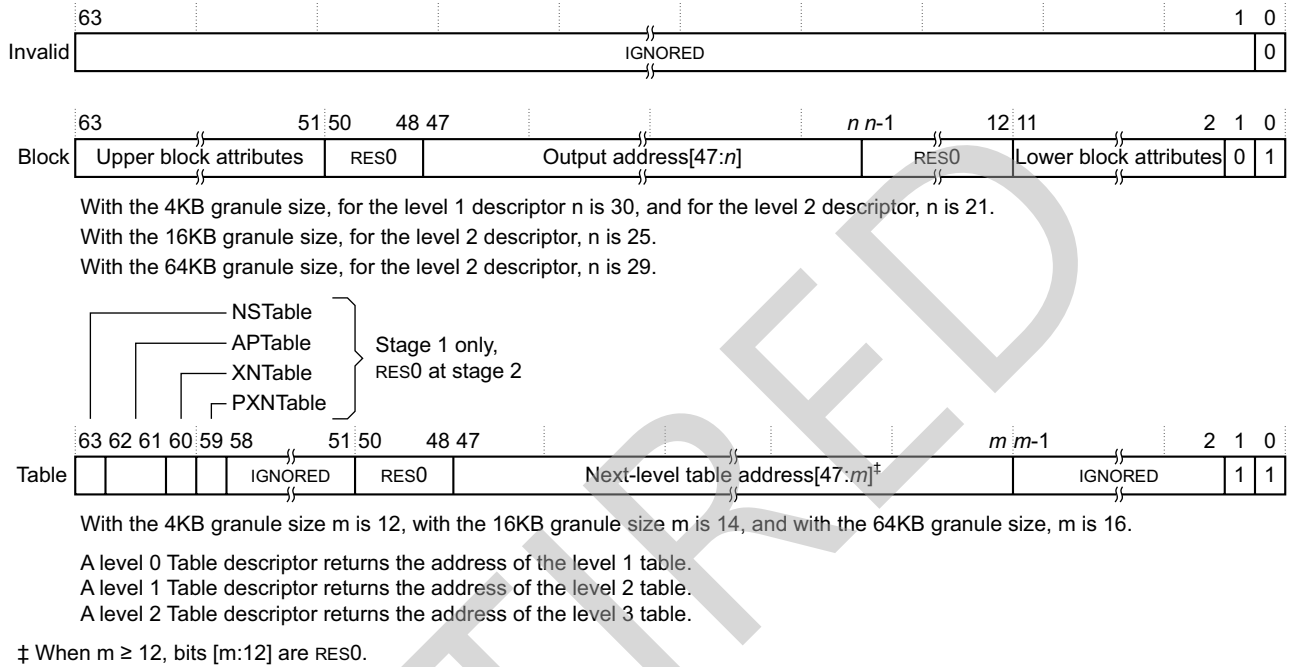


Figure B4-1 VMSAv8-64 level 0, level 1, and level 2 descriptor formats

Figure B4-2 shows the ARMv8.1 level 3 descriptor formats. Bits[63:51] of the descriptor define the memory attributes, and bits[50:48] are RES0.

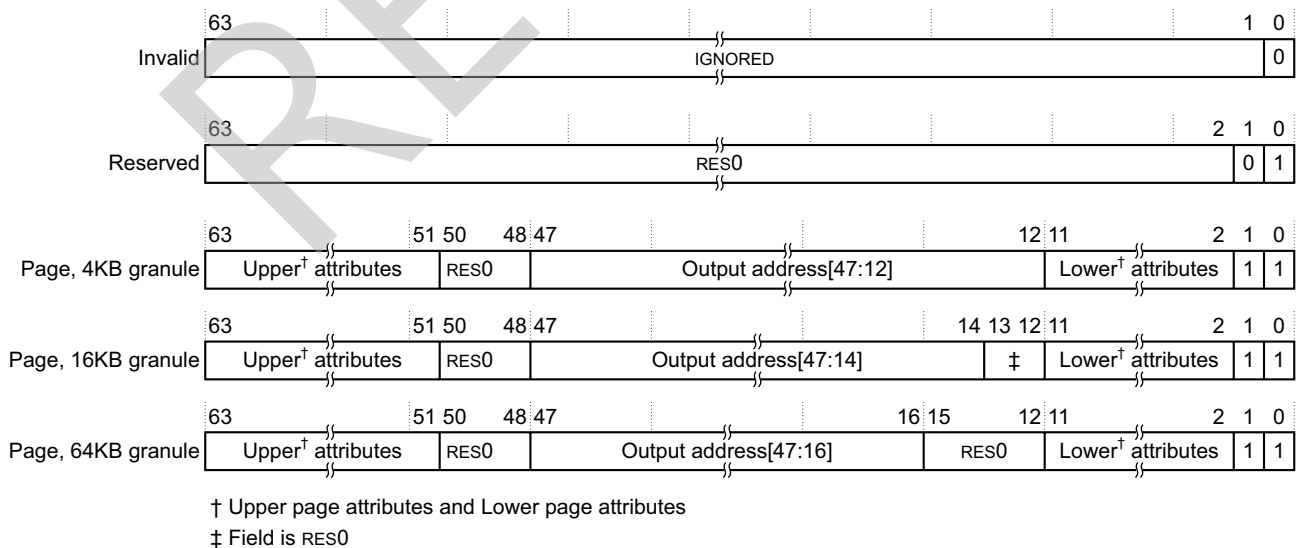
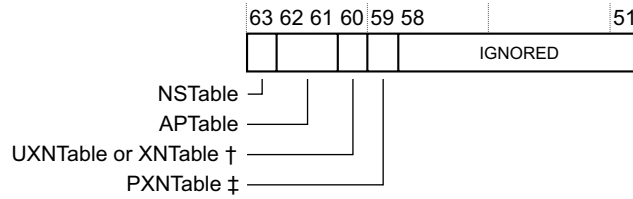


Figure B4-2 VMSAv8-64 level 3 descriptor format

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the attributes for the next-level translation table access, and bits[58:51] are IGNORED.

Next-level descriptor attributes, stage 1 only



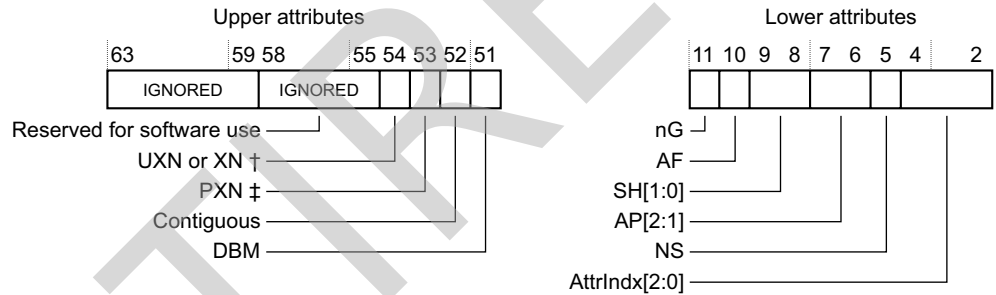
† UXN for a translation regime that applies to execution at EL0, XN for the other regimes.
‡ RES0 for a translation regime that does not apply to execution at EL0.

Figure B4-3 Next-level attributes in stage 1 VMSAv8-64 Table descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block. DBM, bit[51], in the translation table descriptors is the control bit for hardware management of dirty state. The definition of the other attributes is unchanged.

Figure B4-4 shows the attributes for stage 1 Block and Page descriptors.

Attribute fields for VMSAv8-64 stage 1 Block and Page descriptors



† UXN for a translation regime that applies to execution at EL0, XN for the other regimes.
‡ RES0 for a translation regime that does not apply to execution at EL0.

Figure B4-4 Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors

Figure B4-5 shows the attributes for stage 2 Block and Page descriptors.

Attribute fields for VMSAv8-64 stage 2 Block and Page descriptors

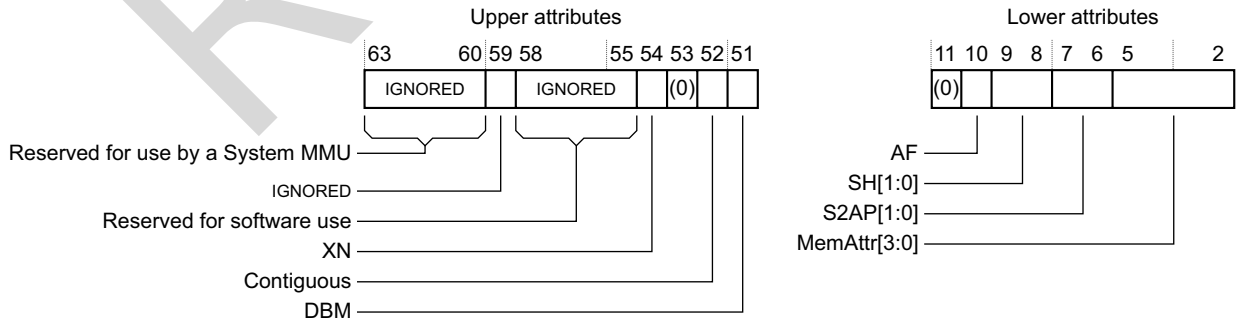


Figure B4-5 Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors

B4.1.2 Hardware management of the Access flag

Hardware management of the Access flag is enabled, for the corresponding stage of address translation, by the following configuration fields:

For stage 1 translations

- `TCR_EL1.HA`.
- `TCR_EL2.HA`.
- `TCR_EL3.HA`.

For stage 2 translations

- `VTCR_EL2.HA`.

When the value of a configuration bit, HA, is 1, then when a memory access is made using a translation table Block or Page descriptor from the corresponding stage of address translation:

- The PE sets the value of the Access flag to 1 in the translation table descriptor in memory, in a coherent manner, by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
 - The descriptor does not generate a Permission fault or an Alignment fault based on the memory type.
 - If the hardware update mechanism was disabled or not implemented, the access would have generated an Access flag fault.

When the PE updates the Access flag in this way no Access flag fault is generated.

- It is CONSTRAINED UNPREDICTABLE whether the PE sets the value of the Access flag in the translation table entry in memory to 1, in a coherent manner, by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
 - The descriptor generates a Permission fault or an Alignment fault based on the memory type.
 - If the hardware update mechanism was disabled or not implemented, the access would have generated an Access flag fault.

This means that the value of the Access flag becomes UNKNOWN if the above conditions are all true.

The Access flag might be set to 1 as a result of speculative accesses by the PE.

———— Note ————

A consequence of the architectural rules for translation table accesses is that the architecture requires that for any translation to which an architecturally executed memory access occurs, the Access flag is set to 1, except as indicated in [Using break-before-make when updating translation table entries on page B4-45](#). However, because the architecture permits speculative accesses, the Access flag is permitted to be set to 1, even if there is no architecturally executed memory accesses by the processor.

When hardware updating of the Access flag is enabled, each stage of translation is treated independently. This means that a single memory access can cause a hardware update to either or both:

- The stage 1 Access flag.
- The stage 2 Access flag.

———— Note ————

Since speculative accesses are permitted to update the Access flags, it is permissible for:

- The stage 1 Access flag for a translation of a virtual address to be updated in situations where the stage 2 translation of the associated intermediate physical address that is returned by the stage 1 of the virtual address does not permit access.
- The stage 2 Access flag for a translation of an intermediate physical address to be updated in situations where the stage 1 translation of the associated virtual address which returned that intermediate physical address does not permit access.

An address translation instruction for an address is not required to set the Access flag in the translation table entries for that address. However, the architecture permits the Access flag in the translation table entries to be set to 1 speculatively.

When hardware updates of the Access flag are enabled for a stage of translation an address translation instruction that uses that stage of translation will not report that the address will give rise to an Access flag fault in the PAR, and the result in PAR will be as if the value of the Access flag in the translation table entries for that address was 1.

Implementations are not required to support the hardware management of the Access flag. If this mechanism is not supported, then the HA bit in [TCR_EL1](#), [TCR_EL2](#), [TCR_EL3](#), and [VTCR_EL2](#) is RES0.

B4.1.3 Hardware management of dirty state

The hardware management of dirty state mechanism can only be enabled if hardware management of the Access flag is enabled.

————— **Note** —————

The hardware management of dirty state mechanism uses:

- In a stage 1 translation table access, the AP[2] bit in conjunction with the DBM bit in the translation table descriptors.
- In a stage 2 translation table access, the S2AP[1] bit in conjunction with the DBM bit in the translation table descriptors.

Hardware management of dirty state is enabled, for the corresponding stage of address translation, by the following configuration fields:

For stage 1 translations

- [TCR_EL1.HD](#).
- [TCR_EL2.HD](#).
- [TCR_EL3.HD](#).

For stage 2 translations

- [VTCR_EL2.HD](#).

When hardware management of dirty state is enabled, and a memory access is made using a translation table Block or Page descriptor:

- For a stage 1 address translation, if the value of the [TCR_ELx.HD](#) field corresponding to the address translation is 1, then the PE sets AP[2] to 0 in the translation descriptor in memory, in a coherent manner by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
 - The value of the DBM field in the descriptor is 1.
 - If the hardware update mechanism was disabled or not implemented, the access using this descriptor would have generated a Permission fault only because the value of the AP[2] field is 1, indicating that the access does not have write permission.

When the PE updates AP[2] in this way no Permission fault is generated because of the value of the AP[2] field.

- For a stage 2 address translation, if the value of the [VTCR_EL2.HD](#) field is 1, then the PE sets S2AP[1] to 1 in the translation descriptor in memory, in a coherent manner by an atomic read-modify-write of the translation table descriptor, if both of the following conditions are true:
 - The value of the DBM field in the descriptor is 1.
 - If the hardware update mechanism was disabled or not implemented, the access using this descriptor would have generated a Permission fault only because the value of the S2AP[1] field is 0, indicating that the access does not have write permission.

When the PE updates S2AP[1] in this way no Permission fault is generated because of the value of the S2AP[1] field.

Note

The PE that does the atomic update of the translation table descriptor is expected to ensure that any cached copy of that translation table descriptor for that PE is similarly updated, or removed from the TLB, so that multiple writes from the same thread on the same PE do not lead to multiple updates to the table. This is only a performance expectation.

If, for a write access, the PE finds that a cached copy of the descriptor in a TLB had the DBM bit set to 1 and the AP[2] or S2AP[1] bit set to the value that forbids writes, then the PE must check that the cached copy is not stale with regard to the descriptor entry in memory, and if necessary perform an atomic read-modify-write update of the descriptor in memory. This applies if the cached copy of the descriptor in a TLB is either:

- A stage 1 descriptor in which DBM has the value 1 and AP[2] has the value 1.
- A stage 2 descriptor in which DBM has the value 1 and S2AP[1] has the value 0.

Note

ARM expects that, in many implementations, any atomic update of a translation table entry required by the dirty state management mechanism will cause a translation table walk.

Implementations are not required to support the dirty state mechanism. If this mechanism is not supported, then the HD bit in [TCR_EL1](#), [TCR_EL2](#), [TCR_EL3](#), and [VTCR_EL2](#) is RES0.

For the hardware updating of the AP[2] and S2AP[1] bits, each translation stage is treated independently. This means a single memory access can update either or both of:

- The stage 1 AP[2] bit.
- The stage 2 S2AP[1] bit.

The architecture does not permit updates to AP[2] and S2AP[1] by the hardware management of dirty state mechanism to occur as a result of speculative accesses by the PE that are not performed architecturally, except that:

- A non-speculative access that passes its stage 1 permissions check can update AP[2] and subsequently encounter a stage 2 fault.

Note

This update of AP[2] is permitted even if the transaction subsequently encounters a stage 2 Translation fault or a Permission fault. This avoids a need to update both AP[2] and S2AP[1] as a single atomic update.

- If the stage 2 hardware management of dirty state mechanism is enabled, the S2AP[1] field of a stage 2 translation table entry that is translating a stage 1 translation table:
 - Is updated from 0 to 1 as a result of a speculative update of the Access flag in an entry of that stage 1 translation table, even though speculative updates of S2AP[1] are not permitted.
 - Without generating a stage 2 MMU fault, is permitted to be updated speculatively from 0 to 1 as a result of performing a translation table walk using that stage 1 translation table, even if the entry in the stage 1 translation table is not updated.

Note

This applies even if the stage 1 translation table contains entries that are not the final level entries and therefore would not be updated. This relaxation avoids the hardware complexity of having to detect whether the stage 1 entry is a final level entry before deciding to set the stage 2 dirty state information.

- If an instruction that generates more than one single-copy atomic memory access has a fault on some, but not all, of those memory accesses, then AP[2] and S2AP[1] bits associated with accesses from that instruction, which do not fault are permitted to be updated if the associated hardware update of dirty state mechanism is enabled.

For a Block or Page translation table descriptor for which the AF bit is 0, the DBM bit is 1, and either the value of the stage 1 AP[2] bit is 1 or the value of the stage 2 S2AP[1] bit is 0, both AF can be set to 1, and either AP[2] set to 0 or S2AP[1] set to 1, in a single atomic read-modify-write operation, as a result of an attempted write to a memory location that uses the translation table entry.

Implications of enabling the dirty state management mechanism

This subsection describes behaviors that result from having the dirty state management mechanism enabled for a particular stage of address translation.

For the final level of lookup in a stage 1 translation:

In the EL3 translation regime

The output address of the lookup is treated as writable if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2] is 1.
- In the descriptor for every higher level of lookup, the value of APTable[1] is 0.

In addition, if these conditions apply and the value of SCTLR_EL3.WXN is 1, then the output address is treated as Execute-never.

In the EL2 translation regime, when the value of HCR_EL2.{E2H, TGE} is not {1, 1}

The output address of the lookup is treated as writable if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2] is 1.
- In the descriptor for every higher level of lookup the value of APTable[1] is 0.

In addition, if these conditions apply and the value of SCTLR_EL2.WXN is 1, then the output address is treated as Execute-never.

In the EL2&0 translation regime, when the value of HCR_EL2.{E2H, TGE} is {1, 1}

The output address of the lookup is treated as writable at EL2 and EL0, Privileged execute-never, but not Unprivileged execute-never, if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is 0b11.
- In the descriptor for every higher level of lookup the value of APTable[1:0] is 0b00.

The output address of the lookup is treated as writable at EL2 but not writable at EL0 if either:

- Both:
 - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is 0b10.
 - In the descriptor for every higher level of lookup the value of APTable[1:0] is 0b0x.In this case, if the value of SCTLR_EL2.WXN is 1 then the output address is treated as Privileged execute-never and Unprivileged execute-never.
- Or both:
 - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is 0b11.
 - In at least one of the descriptors for higher levels of lookup the value of APTable[1:0] is 0b01.

In this case, if the value of SCTLR_EL2.WXN is 1, then the output address is treated as Privileged execute-never and Unprivileged execute-never.

In the EL1&0 translation regime

The output address of the lookup is treated as writable at EL1 and EL0, Privileged execute-never, but not Unprivileged execute-never, if all of the following conditions apply:

- In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is 0b11.

- In the descriptor for every higher level of lookup the value of APTable[1:0] is 0b00.

The output address of the lookup is treated as writable at EL1 but not writable at EL0 if either:

- Both:
 - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is 0b11.
 - In at least one of the descriptors for higher levels of lookup the value of APTable[1:0] is 0b01.

In this case, if the value of [SCTLR_EL1.WXN](#) is 1, then the output address is treated as Privileged execute-never and Unprivileged execute-never.

- Or both:
 - In the descriptor for the final level of lookup, the value of DBM is 1 and the value of AP[2:1] is 0b10.
 - In the descriptor for every higher level of lookup the value of APTable[1:0] is 0b0x.
- In this case, if the value of [SCTLR_EL1.WXN](#) is 1, then the output address is treated as Privileged execute-never and as Unprivileged execute-never.

The output address of a translation table entry where the DBM bit is 1, and the stage 1 AP[2] bit is 1 or the stage 2 S2AP[1] bit is 0, is treated as writable:

- For data cache invalidation instructions that require write permission, that is for the DCIVAC instruction.
- For address translation instructions that require write permission, that is for the AT S12E0W, AT S12E1W, AT S1E0W, AT S1E1W, AT S1E2W, and AT S1E3W instructions.

Cache invalidation and address translation instructions never cause the stage 1 AP[2] bit or the stage 2 S2AP[1] bit in the translation table entry to be updated.

For a Store-Exclusive instruction to a memory location for which the DBM bit is 1 and the stage 1 AP[2] bit is 1, if the Store-Exclusive fails because the exclusive monitor is not in the exclusive state, it is IMPLEMENTATION DEFINED whether the AP[2] bit in the translation table is updated.

For a Store-Exclusive instruction to a memory location for which the DBM bit is 1, and the stage 2 S2AP[1] bit is 0, if the Store-Exclusive fails because the exclusive monitor is not in the exclusive state, it is IMPLEMENTATION DEFINED whether the S2AP[1] bit in the translation table is updated.

For a store to a memory location for which the DBM bit is 1, and the stage 1 AP[2] bit is 1, it is IMPLEMENTATION DEFINED whether the AP[2] bit in the translation table is updated:

- If the memory location generates a synchronous external abort on a write for a store to a memory location.
- If the memory location generates a watchpoint on a write.

For a store to a memory location for which the DBM bit is 1, and the stage 2 S2AP[1] bit is 0, it is IMPLEMENTATION DEFINED whether the S2AP[1] bit in the translation table is updated:

- If the memory location generates a synchronous external abort on a write for a store to a memory location.
- If the memory location generates a watchpoint on a write.

In the event of a PE setting the stage 1 AP[2] bit to 0, it is not required that all associated entries are removed from the TLBs of other PEs in the system.

In the event of a PE setting the stage 2 S2AP[1] bit to 1, it is not required that all associated entries are removed from the TLBs of other PEs in the system.

For the stage 2 translation tables, it is CONSTRAINED UNPREDICTABLE whether the stage 2 S2AP[1] entry is updated in response to a stage 1 translation table walk where the stage 1 translation system is configured to perform hardware updates to the Access flag or stage 1 AP[2] bit, but the values of the Access flag and AP[2] bit are such that a hardware update to the stage 1 translation table entry being accessed is not required.

In the event of a PE encountering a situation for a data write for which the DBM bit is 1 and the stage 1 AP[2] bit is 1 in a TLB, it is required that the hardware checks that the cached copy is not stale with regards to the translation table entry in memory and performs the atomic read-modify-write update with respect to table entry in memory.

In the event of a PE encountering a situation for a data write for which the DBM bit is 1 and stage 2 S2AP[1] bit is 0 in a TLB, it is required that the hardware checks that the cached copy is not stale with regards to the translation table entry in memory and performs the atomic read-modify-write update with respect to table entry in memory.

For a CAS or CASP instruction to a memory location for which the DBM bit is 1, and the stage 1 AP[2] bit is 1, if the compare fails, and the location is not updated, it is CONSTRAINED UNPREDICTABLE whether the AP[2] bit in the translation table is updated.

For a CAS or CASP instruction to a memory location for which the DBM bit is 1, and the stage 2 S2AP[1] bit is 0, if the compare fails, and the location is not updated, it is CONSTRAINED UNPREDICTABLE whether the S2AP[1] bit in the translation table is updated.

B4.1.4 Ordering of hardware updates to the translation tables

A hardware update to the translation table that is caused by a load or a store, including an atomic instruction, is guaranteed to be observed, to the extent required by the shareability attributes:

- Before a load or store, including an atomic instruction, to an arbitrary address, other than the address of the translation table entry, that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load or store.
- Before a load to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load.
- Before a store or atomic access to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry.
- Before a cache maintenance instruction to an arbitrary address appearing in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction that caused the update to the translation table entry and the subsequent cache maintenance instruction.

An update to the translation table that is caused by a load is not ordered with respect to the load itself.

An update to the translation table that is caused by a store or an atomic access is observed by all observers, to the extent required by the shareability attributes, before the store itself in the case that the store is to the same location as the translation table update.

An update to the translation table that is caused by a store or an atomic access is not ordered with respect to the store itself in the case that the store is not the same location as the translation table update.

B4.1.5 Restriction on memory types for hardware updates on page tables

Translation tables can be placed in Normal memory with any cacheability, but the hardware updates to the translation tables require an atomic update of memory. The properties of the atomicity can be met only by functionality outside the PE. Some system implementations might not implement this functionality for all regions of memory. This can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does not support hardware cache coherency.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the hardware updates of the translation tables will be atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

If the hardware updates of the translation tables are not atomic in regard to other agents that access memory, then performing a hardware update to such a location can have one or more of the following effects:

- The hardware update generates a Synchronous external abort, which is presented as an external abort on a translation table walk.
- The instruction generates a System Error interrupt.
- The hardware update generates an Unsupported atomic hardware update MMU fault reported using the new Fault Status code of:
 - ESR_ELx.DFSC = 110001 for Data Aborts.
 - ESR_ELx.IFSC = 110001 for Instruction Aborts.

For the Non-secure EL1&0 translation regime, if atomic hardware update is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.
- The hardware updates are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a system error interrupt.

B4.1.6 Use of the Contiguous bit with hardware updates of the translation table entries

The hardware update of the Access flag, and the AP[2]/S2AP[1] bit only apply to a single translation table entry. An update to one of these bits in a translation table entry that also has the Contiguous bit set to 1 can give rise to translation table entries that have different Access flag, or different AP[2] and S2AP[1] bits within the members of a group of contiguous translation table entries.

This is acceptable under the architecture when using hardware updates of the translation table entries. In addition, an access or a write to a location translated by an entry that has the contiguous bit set might not result in a hardware update of the Access flag or the AP[2]/S2AP[1] bit, if at least one entry in the set of contiguous translation table entries has the Access flag set to 1, or the AP[2]/S2AP[1] bit indicating that the entry is dirty.

Note

- The provision of the Contiguous bit permits, but does not require, the hardware to hold a single entry in a TLB for the set of translation table entries in the group, and to have updated only one or more of the Access flags and the AP[2] bit or S2AP[1] bit for the single translation table entry that gave rise to the TLB entry.
- A consequence of this is that software must combine the Access flag values, and AP[2] or S2AP[1] values, across all translation table entries in a contiguous group to determine whether any of the entries have been accessed or written to.

B4.1.7 Using break-before-make when updating translation table entries

To avoid the effects of TLB caching possibly breaking coherency, ordering guarantees or uniprocessor semantics, failures associated with the hardware updates of the translation tables, or possibly failing to clear the exclusive monitors, the ARM architecture requires the use of a break-before-make when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- A change of the memory type.

- A change of the cacheability attributes.
- A change of the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable.
- A change to the size of block used by the translation system. This applies both:
 - When changing from a smaller size to a larger size, for example by replacing a table mapping with a block mapping in a stage 2 translation table.
 - When changing from a larger size to a smaller size, for example by replacing a block mapping with a table mapping in a stage 2 translation table.
- Creating a global entry when there might be non-global entries in a TLB that overlap with that global entry.

A break-before-make approach on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a DSB instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a DSB instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a DSB instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution. This means the problems described at the start of this subsection cannot arise.

In ARMv8.1, with the introduction of hardware updates to the translation table entries, the effects of not following the break-before-make rules are extended.

If the break-before-make rules are not followed for changing the translation table entries, the ARMv8.1 architecture permits that the following failures associated with the hardware updates of the translation table entries could occur:

- The Access flag is not set on such a translation table entry despite the fact that the memory location associated with that entry was accessed.
- The AP[2] or S2AP[1] bit is modified by the hardware on such a translation table entry despite the fact that the memory location associated with that entry was not written to.
- The AP[2] or S2AP[1] bit is not modified by the hardware on such a translation table entry despite the fact that the memory location associated with that entry was written to.
- The ordering required between hardware updates to such a translation table entry and stores appearing later in program order is not followed.

B4.1.8 Identification mechanism

The [ID_AA64MMFR1_EL1](#).HAFDBS field identifies the support for the hardware management of the Access flag and dirty state.

B4.1.9 See also

In this supplement

- [ID_AA64MMFR1_EL1](#).HAFDBS.
- [TCR_EL1](#).{HD, HA}.
- [TCR_EL2](#).{HD, HA}.
- [TCR_EL3](#).{HD, HA}.
- [VTCR_EL2](#).{HD, HA}.

In the ARM Architecture Reference Manual

The following sections in The AArch64 Virtual Memory System Architecture chapter of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- VMSAv8-64 translation table format descriptors.
- Access controls and memory region attributes.
- Using break-before-make when updating translation table entries.

RETIRED

RETIRED

Chapter B5

AArch64 Privileged Access Never

This chapter describes the addition of a Privileged access never field to PSTATE. It contains the following section:

- [About the Privileged Access Never \(PAN\) bit on page B5-50.](#)

B5.1 About the Privileged Access Never (PAN) bit

A new PAN (Privileged Access Never) state bit is added to PSTATE. When the value of this bit is 1, any access from EL1 or higher to a memory address that is accessible at EL0 generates a Permission fault. A corresponding bit is added to [SPSR_EL1](#), [SPSR_EL2](#), [SPSR_EL3](#) for exception returns, and [DSPSR_EL0](#) for entry to or exit from Debug state.

A new SPAN bit is added to [SCTLR_EL1](#) and [SCTLR_EL2](#), and applies when [HCR_EL2](#).{E2H, TGE} == {1, 1}. The bit is used to control whether the PAN bit is set on an exception to EL1 or EL2.

When the value of the PAN bit is 0, the translation system is the same as in ARMv8.0.

The PAN bit has no effect on:

- Data Cache instructions other than DC ZVA.
- Address translation instructions.
- Unprivileged instructions, LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, STTR, STTRB, and STTRH.
- Instruction accesses.

If access is disabled, then the access will give rise to a stage 1 Permission fault.

On an exception taken from AArch64 to AArch64, PSTATE.PAN is copied to SPSR_ELx.PAN.

On an exception return from AArch64:

- SPSR_ELx.PAN is copied to PSTATE.PAN, when the target Exception level is in AArch64.
- SPSR_ELx.PAN is copied to PSTATE.PAN, when the target Exception level is in AArch32.

PSTATE.PAN is copied to [DSPSR_EL0](#).PAN on entry to Debug state.

[DSPSR_EL0](#).PAN is copied to PSTATE.PAN on exit from Debug state.

B5.1.1 Behavior in Debug state

In Debug state, the behavior of instructions for accessing PSTATE.PAN bit is CONSTRAINED UNPREDICTABLE, and have one of the following behaviors:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction executes as in Non-debug state, setting [DSPSR_EL0](#) and [DLR_EL0](#) to UNKNOWN values.

B5.1.2 Identification mechanism

The [ID_AA64MMFR1_EL1](#).PAN field identifies the support for the Privileged Access Never bit.

B5.1.3 See also

In this supplement

- Instructions:
 - [MSR \(immediate\)](#).
- Registers:
 - [ID_AA64MMFR1_EL1](#).PAN.
 - [SCTLR_EL1](#).SPAN.
 - [SPSR_abt](#).SPAN.
 - [SPSR_EL1](#).PAN.
 - [SPSR_EL2](#).PAN.
 - [SPSR_EL3](#).PAN.
 - [SPSR_fiq](#).SPAN.

- [SPSR_irq.SPAN](#).
- [SPSR_und.SPAN](#).
- [DPSR_EL0.PAN](#).

In the ARM Architecture Reference Manual

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Process state, PSTATE.
- Saved Program Status Registers (SPSRs).
- Access controls and memory region attributes section in The AArch64 Virtual Memory System Architecture chapter.

RETIRED

RETIRED

Chapter B6

Limited Ordering Regions

This chapter describes the Limited ordering regions (LORegions) feature of ARMv8.1. It contains the following section:

- [About limited ordering regions on page B6-54.](#)

B6.1 About limited ordering regions

Limited ordering regions (LORegions) allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the physical address map as observed by a limited set of observers.

This feature is supported in AArch64 only.

B6.1.1 LoadLOAcquire, StoreLORelease

For each PE, the Non-secure physical memory map is divided into a set of LORegions using a table that is held within the PE. Any physical address in the Non-secure memory map can be a member of one LORegion. If a physical address is assigned to more than one LORegion, then an implementation might treat it as if it has been assigned to fewer LORegions than that have been specified. A physical address in the Secure physical memory map cannot be a member of any LORegion.

ARMv8.1 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores that apply in relation to the defined LORegions. The new variants of the load-acquire and store-release instructions are LoadLOAcquire and StoreLORelease.

For all memory types, these new instructions have the following ordering requirements:

- A StoreLORelease followed by a LoadLOAcquire executed on the same PE, where the StoreLORelease and LoadLOAcquire access memory addresses in the same LORegion, is observed in program order by any observers that are in both:
 - The shareability domain of the address that is accessed by the StoreLORelease.
 - The shareability domain of the address that is accessed by the LoadLOAcquire.
- For a LoadLOAcquire, observers in the shareability domain of the address that is accessed by the LoadLOAcquire observe accesses in the following order:
 1. The read caused by the LoadLOAcquire.
 2. Reads and writes caused by loads and stores that appear in program order after the LoadLOAcquire to the same LORegion as the address accessed by the LoadLOAcquire, to the extent required by the Shareability domains of the addresses that are accessed by those loads and stores.

There are no additional ordering requirements on loads or stores that appear before the LoadLOAcquire.

- For a StoreLORelease, observers in the shareability domain of the address that is accessed by the StoreLORelease observe accesses in the following order:
 1. All of the following for which the shareability of the address that is accessed requires that the observer observes the access:
 - Reads and writes to the same LORegion caused by loads and stores that appear in program order before the StoreLORelease.
 - Writes to the same LORegion as the StoreLORelease accesses that were observed by the PE executing the StoreLORelease before it executed the StoreLORelease.
 2. The write caused by the StoreLORelease.

There are no additional ordering requirements on loads or stores that appear in program order after the StoreLORelease.

- A StoreLORelease instruction is multi-copy atomic when observed with a LoadLOAcquire instruction.

In addition, for accesses to a memory-mapped peripheral of an arbitrary system-defined size that is defined using Device memory, these instructions have the following requirements:

- A LoadLOAcquire to an address in the memory-mapped peripheral ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that lie in the same LORegion that are architecturally required to be observed after the LoadLOAcquire will arrive at the memory-mapped peripheral after the memory access of the LoadLOAcquire.

- A StoreLORelease to an address in the memory-mapped peripheral ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that lie in the same LORegion that are architecturally required to be observed before the StoreLORelease will arrive at the memory-mapped peripheral before the memory access of the StoreLORelease.
- Any memory access to the memory-mapped peripheral that are architecturally required to be ordered before the memory access of a StoreLORelease will arrive at the memory-mapped peripheral before any memory access to the same memory-mapped peripheral using Device memory types that are architecturally required to be ordered after the memory access of a LoadLOAcquire to the same memory location as the StoreLORelease, where the LoadLOAcquire has observed the value that is stored by the StoreLORelease.

B6.1.2 Specification of the LORegions

The LORegions are defined in the Non-secure physical memory map using a set of LORegion descriptors. The number of LORegion descriptors is IMPLEMENTATION DEFINED, and can be discovered by reading the LORID_EL1 register.

Each LORegion descriptor consists of:

- A tuple of the following values:
 - A Start Address.
 - An End Address.
 - An LORegion Number.
- Valid bit which indicates whether that LORegion descriptor is valid

A memory location lies within the LORegion identified by the LORegion Number if the physical address lies between the Start Address and the End Address, inclusive. The Start Address must be defined to be aligned to 64KB and the End Address must be defined as the top byte of a 64KB block of memory.

The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1, and LORC_EL1 registers in the System register space. These registers are only supported in the Non-secure memory map.

If a LoadLOAcquire or a StoreLORelease does not match with any LORegion, then:

- The LoadLOAcquire will behave as a LoadAcquire, and will be ordered in the same way with respect to all accesses, independent of their LORegions.
- The StoreLORelease will behave as a StoreRelease, and will be ordered in the same way with respect to all accesses, independent of their LORegions.

Note

If no LORegions are implemented, then the LoadLOAcquire and StoreLORelease will therefore behave as a LoadAcquire and StoreRelease.

B6.1.3 Pseudocode enhancements

A new access type AccType_LIMITEDORDERED has been added for these limited ordering instructions to be identified. This access type is not implemented in the pseudocode, and is just an indication of the memory access.

B6.1.4 Behavior in Debug state

In Debug state, these instructions execute as in Non-debug state.

B6.1.5 Identification mechanism

The ID_AA64MMFR1_EL1.LO field identifies the support for LORegions. The field has the value 0b0001 if the LORegions are supported.

B6.1.6 See also

In this supplement

- Instructions:
 - [LDLAR](#).
 - [LDLARB](#).
 - [LDLARH](#).
 - [STLLR](#).
 - [STLLRB](#).
 - [STLLRH](#).
- Registers:
 - [ID_AA64MMFR1_EL1.LO](#)
 - [HCR_EL2](#).
 - [LORC_EL1](#).
 - [LOREA_EL1](#).
 - [LORID_EL1](#).
 - [LORN_EL1](#).
 - [LORSA_EL1](#).
 - [SCR_EL3](#).

In the ARM Architecture Reference Manual

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Memory barriers section in The AArch64 Application Level Memory Model chapter.
- Loads and stores.
- Load/store exclusive.

Chapter B7

16-bit VMID

This chapter describes the 16-bit VMID feature of ARMv8.1. It contains the following section:

- [VMID size on page B7-58.](#)

B7.1 VMID size

In ARMv8.1, when EL2 is using AArch64, the VMID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

When an implementation supports 16-bit VMID, the [VTTBR_EL2.VMID](#) field contains the 16-bit VMID, and [VTCR_EL2.VS](#) selects whether the top 8 bits of the VMID are used.

When the value of [VTCR_EL2.VS](#) is 1, the top 8 bits of [VTTBR_EL2.VMID](#):

- Are IGNORED by hardware for every purpose other than direct reads of the field.
- Are treated as if they are all zeros when used for allocating and matching TLB entries.

A 16-bit VMID is only supported when EL2 is using AArch64. Hardware must ignore bits[15:9] of a VMID when EL2 is using AArch32.

B7.1.1 Impact on debug

When an implementation has 16 bits of VMID, for context-aware breakpoints that are programmed to match a VMID, [DBGBVR<n>_EL1.VMID](#) and [DBGBXVR<n>.VMID](#) contains the 16-bit VMID.

When an implementation has 8 bits of VMID, the top 8 bits of [DBGBVR<n>_EL1.VMID](#) and [DBGBXVR<n>.VMID](#) of such breakpoints are RES0. See [Breakpoint context comparisons on page B8-69](#).

———— Note ————

- When [VTCR_EL2.VS](#) == 0, or when EL2 is using AArch32.
- To [DBGBXVR<n>\[31:0\]](#) and [DBGBVR<n>_EL1\[63:32\]](#) because matching against the 16 bits of VMID depends on whether EL2 is using AArch64, and not on the Execution state of the debug target Exception level.

B7.1.2 Identification mechanism

The [ID_AA64MMFR1_EL1.VMIDBits](#) field identifies the supported VMID size.

B7.1.3 See also

In this supplement

- [VTCR_EL2.VS](#).
- [VTTBR_EL2.VMID](#).
- [DBGBVR<n>_EL1.VMID](#).
- [DBGBXVR<n>.VMID](#).
- [DBGBVR<n>_EL1.VMID](#) (External).
- [ID_AA64MMFR1_EL1.VMIDBits](#).
- [Breakpoint context comparisons on page B8-69](#).

In the ARM Architecture Reference Manual

The following section of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Virtualization in The AArch64 System Level Programmers' Model chapter.

Chapter B8

Virtualization Host Extensions

This chapter describes the Virtualization Host Extensions feature of ARMv8.1, that add enhanced support for type 2 hypervisors to operation in AArch64 state. It contains the following section:

- [About the Virtualization Host Extensions on page B8-60.](#)

B8.1 About the Virtualization Host Extensions

ARMv8.1 introduces the Virtualization Host Extensions that provide enhanced support for a Type 2 virtualization solution, where there is a Host OS, which is either more privileged than the hypervisor, or is a peer of the hypervisor.

The Virtualization Host Extensions only apply to an implementation that includes EL2 using AArch64.

B8.1.1 State added by the Virtualization Host Extensions

The following state is added as part of the Virtualization Host Extensions:

- A new configuration bit, EL2 host (E2H), is added to [HCR_EL2](#). This only applies to execution in Non-secure state. When [SCR_EL3.NS](#)==0, [HCR_EL2.E2H](#) is treated as 0 for all purposes other than reading or writing the value of the register
- A new 64-bit register, [TTBR1_EL2](#) is added, having the same format and contents as the [TTBR1_EL1](#).
- A new 32-bit register [CONTEXTIDR_EL2](#) is added, having the same format and contents as [CONTEXTIDR_EL1](#).
- A new virtual timer is added to EL2 and is accessed using the registers: [CNTHV_TVAL_EL2](#), [CNTHV_CVAL_EL2](#), and [CNTHV_CVAL_EL2](#). The registers take the same format as [CNTV_CVAL_EL0](#), [CNTV_TVAL_EL0](#), and [CNTV_CTL_EL0](#) respectively. The virtual offset is treated as 0 for this timer.

B8.1.2 Behavior of HCR_EL2.E2H

When the [HCR_EL2.E2H](#) bit is 0:

- There are no changes to the ARMv8 functionality other than the new state described in *State added by the Virtualization Host Extensions*.
- The contents of [TTBR1_EL2](#) are ignored by hardware, other than reads by an MRS instruction and writes by an MSR instruction.
- The Context ID matching breakpoint is disabled at EL2, and uses the value of [CONTEXTIDR_EL1](#) at EL0 and EL1.

When the [HCR_EL2.E2H](#) bit is 1:

- The EL2 translation regime is modified to behave in the same way as the first stage of the EL1&0 translation regime, with an upper address range translated by tables pointed to [TTBR1_EL2](#). The existing [TTBR0_EL2](#) services the lower address range of the EL2 translation regime and is extended to have the same contents and format as the [TTBR0_EL1](#).
- The translation tables used in the EL2 translation regime are modified to take the same format as the EL1&0 translation regime. EL2 accesses are treated as privileged in this format.
- Context ID matching can occur at EL2. When executing at EL2, a Context ID matching breakpoint uses [CONTEXTIDR_EL2](#).
- VMID and VMID + Context ID matching breakpoints do not match at EL2.
- The virtual offset is treated as 0 when [CNTVCT_EL0](#) is read from EL2.
- The Privileged Access Never mechanism applies to accesses from EL2 to a virtual address which has access permitted at EL0.
- The following registers are redefined:
 - [CNTHCTL_EL2](#).
 - [CPTR_EL2](#).
 - [TCR_EL2](#).

If [HCR_EL2](#).{E2H, TGE}== {1, 0}, then all accesses from EL1 and EL0 are not included in the EL2 translation regime.

If `HCR_EL2.{E2H, TGE}` == {1, 1}:

- The EL2&0 translation regime is used when executing at Non-secure EL0 as well as when executing at EL2, where Non-secure EL0 accesses are treated as unprivileged.

Note

Accesses from Non-secure EL1 are not possible under this configuration.

- In EL2, the unprivileged instructions LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, STTR, STTRB, and STTRH act as if they are executing at Non-secure EL0 for permission and watchpoint checking.
- Except for the purpose of reading the value held in the register, some fields in `HCR_EL2` and all fields in `HSTR_EL2` are treated as having a specific value.
- `SCTLR_EL2` is redefined to include additional fields from `SCTLR_EL1`, and to apply to execution at Non-secure EL0.
- The following timer registers are redefined to access the associated `_EL2` register, rather than accessing the `_EL0` register when in EL0:
 - `CNTP_CTL_EL0`.
 - `CNTP_CVAL_EL0`.
 - `CNTP_TVAL_EL0`.
 - `CNTV_CTL_EL0`.
 - `CNTV_CVAL_EL0`.
 - `CNTV_TVAL_EL0`.

For some information on registers that are redirected, see [System and Special-purpose register redirection on page B8-63](#).

- When executing at Non-secure EL0, a Context ID matching breakpoint uses `CONTEXTIDR_EL2`.
- VMID and VMID + Context ID matching breakpoints do not match at Non-secure EL0.
- The `CPACR_EL1` register does not cause any instructions to be trapped to EL1, regardless of the contents of `CPACR_EL1`.
- The `CNTKCTL_EL1` register does not cause any instructions to be trapped to EL1, and the event stream event caused by the `CNTKCTL_EL1` is disabled, regardless of the contents of `CNTKCTL_EL1`.
- The virtual offset is treated as 0 when `CNTVCT_EL0` is read from EL0 or EL2.
- The TLB maintenance and address translation instructions that apply to the EL1&0 translation regime are redefined to apply to the EL2&0 translation regime. See [System instructions on page B12-504](#).
- When executing at EL2 or Non-secure EL0, any physical interrupt that is configured to be taken at EL2 is subject to the Process state interrupt mask. If the mask bit is set, then the corresponding interrupt will not be taken. If the mask bit is not set, then the corresponding interrupt will be taken. See [Asynchronous exception masking on page B8-62](#).
- When an exception is taken from EL0 to EL2, the value of the `HCR_EL2.RW` bit is not considered when determining the exception vector offset to use.

Table B8-1 lists the vector offsets used when an exception is taken from EL0.

Table B8-1 Vector offsets from vector table base address

Exception taken from	Offset for exception type			
	Synchronous	IRQ or vIRQ	FIQ or vFIQ	SError or vSError
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64. ^a	0x400	0x480	0x500	0x580
Lower Exception level, where the implemented level immediately lower than the target level is using AArch32. ^a	0x600	0x680	0x700	0x780

- a. For exceptions taken to EL3, if EL2 is implemented, the level immediately lower than the target level is EL2 if the exception was taken from Non-secure state, but EL1 if the exception was taken from Secure EL1 or EL0.

B8.1.3 Asynchronous exception masking

The following tables show the masking of physical interrupts when the highest implemented Exception level is using AArch64:

- For implementations that include both EL2 and EL3, see Table B8-2.
- For implementations that include EL2 but not EL3, see Table B8-3 on page B8-63.
- Virtual interrupt masking, see Table B8-4 on page B8-63.

In the tables:

- A** When the interrupt is asserted it is taken regardless of the value of the Process state interrupt mask.
- B** When the interrupt is asserted it is subject to the corresponding Process state mask. If the value of the mask is 1 then the interrupt is not taken. If the value of the mask is 0 the interrupt is taken.
- C** When the interrupt is asserted it is not taken, regardless of the value of the Process state interrupt mask.

Table B8-2 Physical interrupt masking when both EL3 and EL2 are implemented

SCR_EL3.EA SCR_EL3.IRQ SCR_EL3.FIQ	SCR_EL3.RW	AMO ^a IMO ^a FMO ^a	Target Exception level	Effect of the interrupt mask when executing in:					
				Non-secure			Secure		
				EL0	EL1	EL2	EL0	EL1	EL3
0	0	0	EL1	B	B	B	B	B	C
		1	EL2	A	A	B	B	B	C
	1	0	EL1	B	B	C	B	B	C
		1	EL2	A	A	B	B	B	C
1	X	X	EL3	A	A	A	A	A	B

- a. If EL2 is using AArch64, these are the HCR_EL2.{AMO, IMO, FMO} control bits. If HCR_EL2.{E2H, TGE} is {0, 1}, these bits are treated as being 1 other than a direct read. If HCR_EL2.{E2H, TGE} is {1, 1}, these bits are treated as being 0 other than a direct read. If EL2 is using AArch32, these are the HCR{AMO, IMO, FMO} control bits. If HCR.TGE is 1, these bits are treated as being 1 other than a direct read.

Table B8-3 Physical interrupt masking when EL3 is not implemented and EL2 is implemented

HCR_EL2.AMO^a HCR_EL2.IMO^a HCR_EL2.FMO^a	Target Exception level	Effect of the interrupt mask when executing in:		
		Non-secure		
		EL0	EL1	EL2
0	EL1	B	B	C
1	EL2	A	A	B

- a. If HCR_EL2.{E2H, TGE} is {0, 1}, these bits are treated as being 1 other than for a direct read.
If HCR_EL2.{E2H, TGE} is {1, 1}, these bits are treated as being 0 other than a direct read.

Table B8-4 Virtual interrupt masking

SCR_EL3.EA SCR_EL3.IRQ SCR_EL3.FIQ	FMO^a IMO^a AMO^a	TGE^a	Effect of the interrupt mask when executing in:					
			Non-secure			Secure		
			EL0	EL1	EL2	EL0	EL1	EL3
X	0	X	C	C	C	C	C	C
X	1	0	B	B	C	C	C	C
X	1	1	C	C	C	C	C	C

- a. If EL2 is using AArch64, these are the HCR_EL2.{TGE, AMO, IMO, FMO} control bits. If EL2 is using AArch32, these are the HCR.{TGE, AMO, IMO, FMO} control bits.

B8.1.4 System and Special-purpose register redirection

When the Virtualization Host Extensions are implemented, and HCR_EL2.E2H is set to 1, when executing at EL2, some EL1 System register access instructions are redefined to access the equivalent EL2 register. Register access behavior is unchanged when executing at EL3, EL1, or EL0.

Table B8-5 shows the System register access instruction encodings that are redirected to the equivalent EL2 register when the named mnemonic is used.

Table B8-5 System register redirection

System register access instruction encoding					Mnemonic	Equivalent register accessed at EL2
op0	op1	CRn	CRm	op2		
3	0	1	0	0	SCTLR_EL1	SCTLR_EL2
				2	CPACR_EL1	CPTR_EL2
		2	0	0	TTBR0_EL1	TTBR0_EL2
				1	TTBR1_EL1	TTBR1_EL2
				2	TCR_EL1	TCR_EL2
		5	1	0	AFSR0_EL1	AFSR0_EL2
				1	AFSR1_EL1	AFSR1_EL2
				2	ESR_EL1	ESR_EL2
		6	0	0	FAR_EL1	FAR_EL2
		10	2	0	MAIR_EL1	MAIR_EL2
				3	AMAIR_EL1	AMAIR_EL2
		12	0	0	VBAR_EL1	VBAR_EL2
		13	0	1	CONTEXTIDR_EL1	CONTEXTIDR_EL2
		14	1	0	CNTKCTL_EL1	CNTHCTL_EL2
3	3	14	2	0	CNTP_TVAL_EL0	CNTHP_TVAL_EL2
				1	CNTP_CTL_EL0	CNTHP_CTL_EL2
				2	CNTP_CVAL_EL0	CNTHP_CVAL_EL2
3	3	14	3	0	CNTV_TVAL_EL0	CNTHV_TVAL_EL2
				1	CNTV_CTL_EL0	CNTHV_CTL_EL2
				2	CNTV_CVAL_EL0	CNTHV_CVAL_EL2

Table B8-6 shows the Special-purpose register access instruction encodings that are redirected to the equivalent EL2 register when the named mnemonic is used.

Table B8-6 Special-purpose register redirection

Special-purpose register access instruction encoding			Mnemonic	Equivalent register accessed at EL2
op1	CRm	op2		
0	0	0	SPSR_EL1	SPSR_EL2
		1	ELR_EL1	ELR_EL2

B8.1.5 System and Special-purpose register aliasing

New register encodings, and aliases, are provided so that software executing at EL2 can access the EL1 registers for which accesses from EL2 are redirected as described in [System and Special-purpose register redirection on page B8-63](#). These aliases can also be used at EL3, but are UNDEFINED at EL1 and EL0.

Table B8-7 shows the System register aliasing.

Table B8-7 System register aliases

System register access instruction encoding					Mnemonic	Register accessed
op0	op1	CRn	CRm	op2		
3	5	1	0	0	SCTLR_EL12	SCTLR_EL1
				2	CPACR_EL12	CPACR_EL1
		2	0	0	TTBR0_EL12	TTBR0_EL1
				1	TTBR1_EL12	TTBR1_EL1
				2	TCR_EL12	TCR_EL1
		5	1	0	AFSR0_EL12	AFSR0_EL1
				1	AFSR1_EL12	AFSR1_EL1
				2	ESR_EL12	ESR_EL1
		6	0	0	FAR_EL12	FAR_EL1
				0	MAIR_EL12	MAIR_EL1
		10	2	0	AMAIR_EL12	AMAIR_EL1
				0	VBAR_EL12	VBAR_EL1
		12	0	0	CONTEXTIDR_EL12	CONTEXTIDR_EL1
				1	CNTKCTL_EL12	CNTKCTL_EL1
		14	1	0	CNTP_TVAL_EL12	CNTP_TVAL_EL0
				1	CNTP_CTL_EL02	CNTP_CTL_EL0
				2	CNTP_CVAL_EL02	CNTHP_CVAL_EL2
3	5	14	3	0	CNTV_TVAL_EL02	CNTV_TVAL_EL0
				1	CNTV_CTL_EL02	CNTV_CTL_EL0
				2	CNTV_CVAL_EL02	CNTHV_CVAL_EL2

Table B8-7 shows the Special-purpose register aliasing.

Table B8-8 Special-purpose register aliases

Special-purpose register access instruction encoding			Register name	Register accessed
op1	CRm	op2		
5	0	0	SPSR_EL12	SPSR_EL1
		1	ELR_EL12	ELR_EL1

B8.1.6 Impact on Debug

For context-aware breakpoints, the Virtualization Host Extensions add new breakpoint types, and modify the existing breakpoint types.

Note

The changes to the breakpoint types apply to the AArch32 registers [DBG BXVR<n>](#) and [DBG BVR<n>](#) because matching against [CONTEXTIDR_EL2](#) depends on whether EL2 is using AArch64, and not on the Execution state of the debug target Exception level.

ARMv8.1 changes to breakpoint types defined by [DBGBCRn_EL1.BT](#)

The following list describes the new and modified breakpoint types introduced in ARMv8.1. The description of the breakpoint types not listed here is unchanged.

[0b0010](#), Unlinked Context ID Match breakpoint

[BT](#) = [0b0010](#) is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- [DBGBCR<n>_EL1](#).{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions for.
- A successful Context ID match, as described in [Breakpoint context comparisons on page B8-69](#).

The value of [DBG BVR<n>_EL1](#)[31:0] is compared with the current Context ID.

[CONTEXTIDR_EL2](#) holds the current Context ID when all of:

- The implementation includes the Virtualization Host Extensions.
- The PE is in Non-secure state.
- [HCR_EL2](#).E2H is set to 1.
- The PE is executing at EL0 and [HCR_EL2](#).TGE is 1, or PE is executing at EL2.

Otherwise, [CONTEXTIDR_EL1](#) holds the current Context ID.

[DBGBCR<n>_EL1](#).{LBN, BAS} for this breakpoint are ignored.

[0b0011](#), Linked Context ID Match breakpoint

[BT](#) = [0b0011](#) is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
 - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint.
 - A successful Context ID match, as described in [Breakpoint context comparisons on page B8-69](#).
- Generation of a Watchpoint exception depends on both:
 - A successful data address match, defined by a Linked watchpoint that links to this breakpoint.
 - A successful Context ID match, as described in [Breakpoint context comparisons on page B8-69](#).

The value of [DBG BVR<n>_EL1](#)[31:0] is compared with the current Context ID.

[CONTEXTIDR_EL2](#) holds the current Context ID when all of:

- The implementation includes the Virtualization Host Extensions.
- The PE is in Non-secure state.
- [HCR_EL2](#).E2H is set to 1.

- The PE is executing at EL0 and [HCR_EL2.TGE](#) is 1, or PE is executing at EL2.

Otherwise, [CONTEXTIDR_EL1](#) holds the current Context ID.

[DBGBCR<n>_EL1](#). {LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b0110, Unlinked [CONTEXTIDR_EL1](#) Match

BT = 0b0110 is a reserved value if the breakpoint is not a context-aware breakpoint, or if the implementation does not include the Virtualization Host Extensions.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- [DBGBCR<n>_EL1](#). {SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons on page B8-69](#).

The Context ID check is made against the value in [CONTEXTIDR_EL1](#). The value of [DBGBVR<n>_EL1](#)[31:0] is compared with the Context ID value held in [CONTEXTIDR_EL1](#).

[DBGBCR<n>_EL1](#). {LBN, BAS} for this breakpoint are ignored.

0b0111, Linked [CONTEXTIDR_EL1](#) Match

BT = 0b0111 is a reserved value if the breakpoint is not a context-aware breakpoint, or if the implementation does not include the Virtualization Host Extensions.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
 - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint.
 - A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons on page B8-69](#).
- Generation of a Watchpoint exception depends on both:
 - A successful data address match, defined by a Linked watchpoint that links to this breakpoint.
 - A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons on page B8-69](#).

The Context ID check is made against the value in [CONTEXTIDR_EL1](#). The value of [DBGBVR<n>_EL1](#)[31:0] is compared with the Context ID value held in [CONTEXTIDR_EL1](#).

[DBGBCR<n>_EL1](#). {LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b1100, Unlinked [CONTEXTIDR_EL2](#) Match

BT = 0b1100 is a reserved value if the breakpoint is not a context-aware breakpoint, or if the implementation does not include the Virtualization Host Extensions.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- [DBGBCR<n>_EL1](#). {SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful [CONTEXTIDR_EL2](#) match, as described in [Breakpoint context comparisons on page B8-69](#).

The Context ID check is made against the value in [CONTEXTIDR_EL2](#). The check against [CONTEXTIDR_EL2](#) means this breakpoint can be generated only if execution is in Non-secure state and EL2 is using AArch64.

The match fails if execution is in Secure state, or if EL2 is using AArch32. Otherwise, the value of [DBGBVR<n>_EL1](#) is compared with the Context ID value held in [CONTEXTIDR_EL2](#).

[DBGBCR<n>_EL1](#). {LBN, BAS} for this breakpoint are ignored.

0b1101, Linked CONTEXTIDR_EL2 Match

BT == 0b1101 is a reserved value if the breakpoint is not a context-aware breakpoint, or if the implementation does not include the Virtualization Host Extensions.

For context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
 - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint.
 - A successful [CONTEXTIDR_EL2](#) match, as described in [Breakpoint context comparisons](#) on page B8-69.
- Generation of a Watchpoint exception depends on both:
 - A successful data address match, defined by a Linked watchpoint that links to this breakpoint.
 - A successful [CONTEXTIDR_EL2](#) match, as described in [Breakpoint context comparisons](#) on page B8-69.

The Context ID check is made against the value in [CONTEXTIDR_EL2](#), regardless of the value of [HCR_EL2.E2H](#). The check against the [CONTEXTIDR_EL2](#) means this breakpoint or watchpoint can be generated only if execution is in Non-secure state and EL2 is using AArch64.

The match fails if execution is in Secure state, or if EL2 is using AArch32. Otherwise, the value of [DBGBVR<n>_EL1](#) is compared with the Context ID value held in [CONTEXTIDR_EL2](#).

[DBGBCR<n>_EL1](#).{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b1110, Unlinked Full Context ID Match

BT == 0b1110 is a reserved value if the breakpoint is not a context-aware breakpoint, or if the implementation does not include the Virtualization Host Extensions.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- [DBGBCR<n>_EL1](#).{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in [Breakpoint context comparisons](#) on page B8-69.

The Context ID check is made by checking the value of [DBGBVR<n>_EL1](#)[31:0] against the value in [CONTEXTIDR_EL1](#) and the value of [DBGBVR<n>_EL1](#)[63:32] against the value in [CONTEXTIDR_EL2](#). Both comparisons must match for the check to succeed.

The check against the [CONTEXTIDR_EL2](#) means this breakpoint can be generated only if execution is in Non-secure state and EL2 is using AArch64.

The match fails if execution is in Secure state, or if EL2 is using AArch32.

[DBGBCR<n>_EL1](#).{LBN, BAS} for this breakpoint are ignored.

0b1111, Linked Full Context ID Match

BT == 0b1111 is a reserved value if the breakpoint is not a context-aware breakpoint, or if the implementation does not include the Virtualization Host Extensions.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
 - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint.
 - A successful Context ID match, as described in [Breakpoint context comparisons](#) on page B8-69.

- Generation of a Watchpoint exception depends on both:
 - A successful data address match, defined by a Linked watchpoint that links to this breakpoint.
 - A successful Context ID match, as described in *Breakpoint context comparisons*.

The Context ID check is made by checking the value of `DBGBVR<n>_EL1[31:0]` against the value in `CONTEXTIDR_EL1` and the value of `DBGBVR<n>_EL1[63:32]` against the value in `CONTEXTIDR_EL2`. Both comparisons must match for the check to succeed.

The check against the `CONTEXTIDR_EL2` means this breakpoint or watchpoint can be generated only if execution is in Non-secure state and EL2 is using AArch64.

`DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC}` for this breakpoint are ignored.

Breakpoint context comparisons

A context comparison is successful if, depending on the breakpoint type set by `DBGBCR<n>_EL1.BT`, one of the following is true:

- The current Context ID value is equal to `DBGBVR<n>_EL1[31:0]`.
- The current VMID value is equal to `DBGBVR<n>_EL1.VMID`.
- `CONTEXTIDR_EL2` is `DBGBVR<n>_EL1[31:0]` and the current VMID value is equal to `DBGBVR<n>_EL1.VMID`.
- `CONTEXTIDR_EL1` is equal to `DBGBVR<n>_EL1[31:0]`.
- `CONTEXTIDR_EL2` is equal to `DBGBVR<n>_EL1[63:32]`.
- `CONTEXTIDR_EL1` is equal to `DBGBVR<n>_EL1[31:0]` and `CONTEXTIDR_EL2` is equal to `DBGBVR<n>_EL1[63:32]`.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of `CONTEXTIDR_EL1` and any of:
 - The PE is executing at EL3 using AArch64.
 - The PE is executing at EL2.
 - The Virtualization Host Extensions are implemented, EL2 is using AArch64, the PE is executing in Non-secure state, and `HCR_EL2.{E2H, TGE} == {1, 1}`.
- The comparison uses the value of `CONTEXTIDR_EL2` and any of:
 - The Virtualization Host Extensions are not implemented.
 - The PE is in Secure state.
 - EL2 is using AArch32.
- The comparison uses the current VMID value and any of:
 - EL2 is not implemented.
 - The PE is in Secure state.
 - The PE is executing at EL2.
 - The Virtualization Host Extensions are implemented, EL2 is using AArch64, the PE is executing in Non-secure state, and `HCR_EL2.{E2H, TGE} == {1, 1}`.

Note

- For all Context breakpoints, `DBGBCR<n>_EL1.BAS` is RES1 and is ignored.
- For Linked Context breakpoints, `DBGBCR<n>_EL1.{LBN, SSC, HMC, PMC}` are RES0 and are ignored.

Reserved DBGBCR<n>_EL1.BT values

Table B8-9 shows when particular DBGBCR<n>_EL1.BT values are reserved.

Table B8-9 Reserved BT values

BT value	Breakpoint type	Reserved
0b001x	Context ID Match	For non context-aware breakpoints.
0b010x	Address Mismatch	In stage 1 of an AArch64 translation regime, or if EDSCR.HDE is 1 and halting is allowed.
0b011x	CONTEXTIDR_EL1 Match	When Virtualization Host Extensions is not implemented and breakpoints are not context-aware.
0b100x	VMID Match	For non context-aware breakpoints, or if EL2 is not implemented.
0b101x	Context ID and VMID Match	
0b110x	CONTEXTIDR_EL2 Match	When Virtualization Host Extensions is not implemented and breakpoints are not context-aware.
0b111x	Full Context ID Match	

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
 - Disabled.
 - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<n>_EL1.
- For a direct or external read of DBGBCR<n>_EL1, if the reserved BT value:
 - Has no function for any execution conditions, the value read back is UNKNOWN.
 - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

B8.1.7 Impact on Performance Monitors

Performance monitoring events, CID_WRITE_RETIRED and TTBR_WRITE_RETIRED are extended to support Virtualization Host Extensions:

0x000B, CID_WRITE_RETIRED, Instruction architecturally executed, condition code check pass, write to CONTEXTIDR

The counter is:

- Incremented as a result of the retirement of an instruction accessing the named register CONTEXTIDR_EL1, even when executing at EL2.
- Not incremented as a result of the retirement of an instruction accessing the named register CONTEXTIDR_EL12.

———— Note ————

The event is defined by the name used to access the register. The counter does not count writes to the named register CONTEXTIDR_EL2.

For more information, see [Exception-related events on page B8-71](#).

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening context synchronization operation, it is CONSTRAINED UNPREDICTABLE whether the first write is counted.

0x001C, TTBR_WRITE_RETIRED, Instruction architecturally executed, condition code check pass, write to TTBR

The counter counts writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and TTBR0 and TTBR1 in AArch32 state. When EL3 is implemented and using AArch32, this includes counting writes to both banked copies of TTBR0 and TTBR1.

The counter is:

- Incremented as a result of the retirement of an instruction accessing the named registers TTBR0_EL1 and TTBR1_EL1.
- Not incremented as a result of the retirement of an instruction accessing the named registers TTBR0_EL12 and TTBR1_EL12.

————— Note —————

The event is defined by the name used to access the register. The counter does not count writes to the named registers:

- TTBR0_EL3 if EL3 is implemented and is using AArch64.
- TTBR0_EL2, TTBR1_EL2, and VTTBR_EL2 if EL2 is implemented and is using AArch64.
- HTTBR and VTTBR if EL2 is implemented and is using AArch32.

For more information, see [Exception-related events](#).

If the PE executes two writes to the same TTBR, without an intervening context synchronization operation, it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR is counted.

The ARMv8 debug architecture describes events counting exceptions that are either:

- Taken locally, that is, taken from EL0 to EL1, or taken to the current Exception level.
- Not taken locally, that is, taken from EL0 or EL1 to EL2, or from EL0, EL1, or EL2 to EL3.

For ARMv8.1, when the Virtualization Host Extensions are included, and when the values of [HCR_EL2](#).{E2H, TGE} is {1, 1}, exceptions that are taken from EL0 to EL2 are classified as exceptions that are taken locally.

Exception-related events

In ARMv8 architecture, the PMU must filter some events related to exceptions and exception handling according to the Exception level from which the exception was taken. These events are:

- Exception taken.
- Instruction architecturally executed, condition code check pass, exception return.
- Instruction architecturally executed, condition code check pass, write to CONTEXTIDR.
- Instruction architecturally executed, condition code check pass, write to translation table base.

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR after the return from the exception.

————— Note —————

Unprivileged software cannot write to CONTEXTIDR.

B8.1.8 Identification mechanism

The [ID_AA64MMFR1_EL1](#).VH field identifies the presence of the Virtualization Host Extensions.

The following fields indicate the presence of the Virtualization Host Extensions for debug, including the changes for the Sample-based Profiling Extension and the Performance Monitors Extension:

- [ID_AA64DFR0_EL1](#).DebugVer.

- [ID_DFR0_EL1](#).{CopSDBG, CopDBG}.

B8.1.9 See also

In this supplement

- [CONTEXTIDR_EL1](#).
- [CONTEXTIDR_EL2](#).
- [HCR_EL2](#).{E2H, TGE}.
- [ID_AA64MMFR1_EL1](#).VH.
- [ID_AA64DFR0_EL1](#).DebugVer.
- [ID_DFR0_EL1](#).{CopSDBG, CopDBG}.
- [TCR_EL2](#).
- [DBGBCR<n>_EL1](#).BT.
- [DBGBVR<n>_EL1](#).
- [DBGBVR<n>_EL1](#)(External).
- [EDDEVARCH](#).ARCHVER.
- [Appendix F1 Notes on Using Debug and Performance Monitors](#).

In the ARM Architecture Reference Manual

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

- Virtualization section in The AArch64 System Level Programmers' Model chapter.
- Breakpoint instruction address comparisons.
- Breakpoint types defined by [DBGBCRn_EL1](#).BT.
- Breakpoint usage constraints.
- Watchpoint data address comparisons.
- Common architectural event numbers.

Chapter B9

AArch64 Performance Monitors Extension

This chapter describes the changes to the Performance Monitors Extension introduced with ARMv8.1. It contains the following section:

- [Changes to the Performance Monitors Extension on page B9-74.](#)

B9.1 Changes to the Performance Monitors Extension

The OPTIONAL Performance Monitors Extension is enhanced to:

- Provide a new control to disable event counting at EL2. A control bit HPMD is added to MDCR_EL2 to prohibit event counting at EL2.
- The event number space is extended to 16 bits to allow additional IMPLEMENTATION DEFINED event types and extend the reserved space for future additions to the architecturally-defined event types.
- In an ARMv8.1 implementation, in addition to the events required by PMUv3, the STALL_FRONTEND and STALL_BACKEND events must be implemented. For more information, see [Required events](#).

B9.1.1 Extended event number space

The event number space is extended to 16 bits, and is defined as:

0x0000-0x003F and 0x4000-0x403F

Common architectural and microarchitectural events, discoverable using PMCEID<n>_EL0.

0x0040-0x00BF and 0x4040-0x40BF

ARM recommended common architectural and microarchitectural events. These are IMPLEMENTATION DEFINED.

0x8000-0x80BF and 0xC000-0xC0BF

Reserved.

All other values

IMPLEMENTATION DEFINED events.

To address this extended number space, the [PMEVTYPER<n>_EL0](#).evtCount is extended to 16 bits.

B9.1.2 Required events

PMUv3 requires that an implementation includes the following common events:

- 0x0000, SW_INCR, Instruction architecturally executed, condition code check pass, software increment.
- 0x0003, L1D_CACHE_REFILL, Attributable Level 1 data cache refill.

Note

Event 0x0003 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0004, L1D_CACHE, Attributable Level 1 data cache access.

Note

Event 0x0004 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0010, BR_MIS_PRED, Mispredicted or not predicted branch speculatively executed.

Note

Event 0x0010 is only required if the implementation includes program-flow prediction. However, ARM recommends that the event is implemented as described in the section Common microarchitectural event numbers in Chapter D5 The Performance Monitors Extension of the ARM Architecture Reference Manual.

- 0x0011, CPU_CYCLES, Cycle.
- 0x0012, BR_PRED, Predictable branch speculatively executed.

———— **Note** ————

Event 0x0012 is only required if the implementation includes program-flow prediction. However, ARM recommends that the event is implemented as described in the section Common microarchitectural event numbers in Chapter D5 The Performance Monitors Extension of the ARM Architecture Reference Manual.

- At least one of:
 - 0x0008, INST_RETIRE, Instruction architecturally executed.
 - 0x001B, INST_SPEC, Operation speculatively executed.

———— **Note** ————

ARM strongly recommends that event 0x0008 is implemented.

- 0x0023, STALL_FRONTEND, No operation issued due to the frontend. In an ARMv8.1 implementation, this event must be implemented.
- 0x0024, STALL_BACKEND, No operation issued due to the backend. In an ARMv8.1 implementation, this event must be implemented.

B9.1.3 Identification mechanism

The ID_AA64DFR0_EL1.PMUVer, and EDDFR.PMUVer fields describe the [PMEVTYPER<n>_EL0.evtCount](#) range.

B9.1.4 See also

In this supplement

- [ID_AA64DFR0_EL1.PMUVer](#).
- [MDCR_EL2.HPMD](#).
- [PMCEID0_EL0](#).
- [PMCEID1_EL0](#).
- [PMCR_EL0](#).
- [PMEVTYPER<n>_EL0.evtCount](#).
- [EDDFR.PMUVer](#).

In the ARM Architecture Reference Manual

The following sections will be updated:

- The Performance Monitors Extension chapter.
- Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events.

RETIRED

Chapter B10

A64 Instruction Set Encoding

This chapter describes the encoding of the instructions that ARMv8.1 adds to the A64 instruction set. It contains the following sections:

- [Loads and stores on page B10-78.](#)
- [Data processing - SIMD and floating point on page B10-81.](#)

B10.1 Loads and stores

The new instructions are added to classes within the Loads and stores instruction group. This section describes the encoding of the new classes and the new instructions added to the Loads and stores instruction group.

The instructions added to the Load/store exclusive instruction class are:

- The CAS and CASP instructions.
- The LDLAR and STLLR instructions.

The LD<OP> and SWP instructions are added to the Atomic memory operations instruction class. This is a new instruction class introduced in ARMv8.1. The encoding space used for this instruction class is UNDEFINED in ARMv8.0.

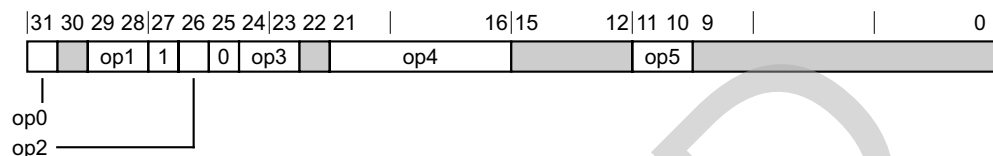
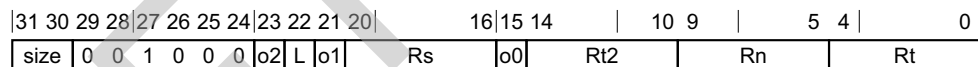


Table B10-1 Encoding table for the Loads and Stores group

Decode fields						Decode group
op0	op1	op2	op3	op4	op5	
-	00	00	0x	-	-	<i>Load/store exclusive</i>
-	11	-	0x	1xxxxx	00	<i>Atomic memory operations on page B10-80</i>

B10.1.1 Load/store exclusive

The section describes the encoding of the new instructions in the Load/store exclusive instruction class. The encodings of the other instructions in this instruction class remain unchanged.



The following table shows the allocation of encodings of the new atomic instructions in the Load/store exclusive instruction class.

Decode fields						Instruction page
size	o2	L	o1	o0	Rt2	
00	0	0	1	0	11111	<i>CASP, CASPA, CASPAL, CASPL - 32-bit, no memory ordering variant</i>
00	0	0	1	1	11111	<i>CASP, CASPA, CASPAL, CASPL - 32-bit, release variant</i>
00	0	1	1	0	11111	<i>CASP, CASPA, CASPAL, CASPL - 32-bit, acquire variant</i>
00	0	1	1	1	11111	<i>CASP, CASPA, CASPAL, CASPL - 32-bit, acquire and release variant</i>
00	1	0	0	0	-	<i>STLLRB</i>
00	1	0	1	0	11111	<i>CASB, CASAB, CASALB, CASLB - No memory ordering variant</i>
00	1	0	1	1	11111	<i>CASB, CASAB, CASALB, CASLB - Release variant</i>
00	1	1	0	0	-	<i>LDLARB</i>

Decode fields						Instruction page
size	o2	L	o1	o0	Rt2	
00	1	1	1	0	11111	CASB, CASAB, CASALB, CASLB - Acquire variant
00	1	1	1	1	11111	CASB, CASAB, CASALB, CASLB - Acquire and release variant
01	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL - 64-bit, no memory ordering variant
01	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL - 64-bit, release variant
01	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL - 64-bit, acquire variant
01	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL - 64-bit, acquire and release variant
01	1	0	0	0	-	STLLRH
01	1	0	1	0	11111	CASH, CASAH, CASALH, CASLH - No memory ordering variant
01	1	0	1	1	11111	CASH, CASAH, CASALH, CASLH - Release variant
01	1	1	0	0	-	LDLARH
01	1	1	1	0	11111	CASH, CASAH, CASALH, CASLH - Acquire variant
01	1	1	1	1	11111	CASH, CASAH, CASALH, CASLH - Acquire and release variant
10	1	0	0	0	-	STLLR - 32-bit variant
10	1	0	1	0	11111	CAS, CASA, CASAL, CASL - 32-bit, no memory ordering variant
10	1	0	1	1	11111	CAS, CASA, CASAL, CASL - 32-bit, release variant
10	1	1	0	0	-	LDLAR - 32-bit variant
10	1	1	1	0	11111	CAS, CASA, CASAL, CASL - 32-bit, acquire variant
10	1	1	1	1	11111	CAS, CASA, CASAL, CASL - 32-bit, acquire and release variant
11	1	0	0	0	-	STLLR - 64-bit variant
11	1	0	1	0	11111	CAS, CASA, CASAL, CASL - 64-bit, no memory ordering variant
11	1	0	1	1	11111	CAS, CASA, CASAL, CASL - 64-bit, release variant
11	1	1	0	0	-	LDLAR - 64-bit variant
11	1	1	1	0	11111	CAS, CASA, CASAL, CASL - 64-bit, acquire variant
11	1	1	1	1	11111	CAS, CASA, CASAL, CASL - 64-bit, acquire and release variant

B10.1.2 Atomic memory operations

The section describes the encoding of the new atomic instructions in the Atomic memory operations instruction class.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		111		V		0		A		R		1		Rs			o3		opc		0		Rn			Rt					

The following table shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Decode fields				Instruction page
size	V	o3	opc	
1x	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL
00	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB
01	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH
1x	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL
00	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB
01	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH
1x	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL
00	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB
01	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH
1x	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL
00	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB
01	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH
1x	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL
00	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB
01	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH
1x	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL
00	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB
01	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH
1x	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL
00	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB
01	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH
1x	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL
00	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB
01	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH
1x	0	1	000	SWP, SWPA, SWPAL, SWPL
00	0	1	000	SWPB, SWPAB, SWPALB, SWPLB
01	0	1	000	SWPH, SWPAH, SWPALH, SWPLH

B10.2 Data processing - SIMD and floating point

The new SIMD instructions are added to the Data processing - SIMD and floating point instruction group.

31	28 27	24 23 22	19 18 17 16 15	10 9		0
op0	111	op1	op2	op3	op4	

The scalar forms of the instructions are grouped under the following instruction classes:

- Advanced SIMD scalar three same extra.
- Advanced SIMD scalar x indexed element.

The vector forms of the instructions are grouped under the following instruction classes:

- Advanced SIMD three same.
- Advanced SIMD vector x indexed element.

The encoding space used for the Advanced SIMD scalar three same extra is UNDEFINED in ARMv8.0.

Table B10-2 Encoding table for the Data processing - Scalar Advanced SIMD and Floating-point group

Decode fields					Decode group
op0	op1	op2	op3	op4	
01x1	0x	x0xx	-	1xxxx1	Advanced SIMD scalar three same extra
01x1	1x	-	-	xxxxx0	Advanced SIMD scalar x indexed element on page B10-82
0xx0	0x	x1xx	-	xxxxx1	Advanced SIMD three same on page B10-82
0xx0	1x	-	-	xxxxx0	Advanced SIMD vector x indexed element on page B10-82

B10.2.1 Advanced SIMD scalar three same extra

The section describes the encoding of the Advanced SIMD scalar three same extra instruction class. The encodings of the other instructions in this instruction class remain unchanged.

31 30 29 28 27 26 25 24 23 22 21 20	16 15	11 10 9	5 4	0
0 1 U 1 1 1 1 0 size 1	Rm	opcode 1	Rn	Rd

Decode fields			Instruction page
U	size	opcode	
1	-	0000	SQRDMLAH (vector) - Scalar on page B11-161
1	-	0001	SQRDMLSH (vector) - Scalar on page B11-166

B10.2.2 Advanced SIMD scalar x indexed element

The section describes the encoding of the Advanced SIMD scalar x indexed element instruction class. The encodings of the other instructions in this instruction class remain unchanged.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	11	10	9	5	4	0
0	1	U	1	1	1	1	0	size	1	Rm	opcode	1	Rn	Rd					

Decode fields

Instruction page

U size opcode

1	-	0000	SQRDMLAH (by element) - <i>Scalar</i> on page B11-161
1	-	0001	SQRDMLSH (by element) - <i>Scalar</i> on page B11-163

B10.2.3 Advanced SIMD three same

This section describes the encoding of the Advanced SIMD three same instruction class. The encodings of the other instructions in this instruction class remain unchanged.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	size	1	Rm	opcode	1	Rn	Rd					

Decode fields

Instruction page

U size opcode

1	-	0000	SQRDMLAH (vector) - <i>Vector</i> on page B11-161
1	-	0001	SQRDMLSH (vector) - <i>Vector</i> on page B11-166

B10.2.4 Advanced SIMD vector x indexed element

This section describes the encoding of the Advanced SIMD vector x indexed element instruction class. The encodings of the other instructions in this instruction class remain unchanged.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	11	10	9	5	4	0
0	Q	U	0	1	1	1	0	size	1	Rm	opcode	1	Rn	Rd					

Decode fields

Instruction page

U size opcode

1	-	0000	SQRDMLAH (by element) - <i>Vector</i> on page B11-158
1	-	0001	SQRDMLSH (by element) - <i>Vector</i> on page B11-163

B10.2.5 See also

In the ARM Architecture Reference Manual

The following sections will be updated:

- Data processing - SIMD and floating point.
- Advanced SIMD scalar x indexed element.
- Advanced SIMD three same.
- Advanced SIMD vector x indexed element.

RETIRED

RETIRED

Chapter B11

A64 Instructions

This chapter describes the new A64 instructions introduced in ARMv8.1. It contains the following section:

- [Alphabetical list of instructions on page B11-86.](#)
- [ARMv8.0 sections relating to these instructions on page B11-227.](#)

B11.1 Alphabetical list of instructions

RETIRED

B11.1.1 CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is stored to memory.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	x	0	0	1	0	0	0	1	L	1		Rs	o0	1	1	1	1	1		Rn		Rt
size																						

32-bit, acquire variant

Applies when size == 10 && L == 1 && o0 == 0.

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

32-bit, acquire and release variant

Applies when size == 10 && L == 1 && o0 == 1.

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

32-bit, no memory ordering variant

Applies when size == 10 && L == 0 && o0 == 0.

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

32-bit, release variant

Applies when size == 10 && L == 0 && o0 == 1.

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

64-bit, acquire variant

Applies when size == 11 && L == 1 && o0 == 0.

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

64-bit, acquire and release variant

Applies when size == 11 && L == 1 && o0 == 1.

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

64-bit, no memory ordering variant

Applies when size == 11 && L == 0 && o0 == 0.

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

64-bit, release variant

Applies when size == 11 && L == 0 && o0 == 1.

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

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

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

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue; // all observers in the shareability domain
                                                         // observe the load and store atomically

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


B11.1.2 CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory loads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is stored to memory.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
0	0	0	0	1	0	0	0	1	L	1		Rs	o0	1	1	1	1	1		Rn		Rt
size																						

Acquire variant

Applies when L == 1 && o0 == 0.

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

Acquire and release variant

Applies when L == 1 && o0 == 1.

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

No memory ordering variant

Applies when L == 0 && o0 == 0.

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

Release variant

Applies when L == 0 && o0 == 1.

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

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

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

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

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue; // all observers in the shareability domain
                                                         // observe the load and store atomically

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

B11.1.3 CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory loads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is stored to memory.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31 30 29 28 27 26 25 24 23 22 21 20												16 15 14 13 12 11 10 9								5 4		0	
0 1		0 0 1 0 0 0				1 L 1		Rs				o0 1 1 1 1		Rn				Rt					
size																							

Acquire variant

Applies when $L == 1 \ \&\& \ o0 == 0$.

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

Acquire and release variant

Applies when $L == 1 \ \&\& \ o0 == 1$.

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

No memory ordering variant

Applies when $L == 0 \ \&\& \ o0 == 0$.

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

Release variant

Applies when $L == 0 \ \&\& \ o0 == 1$.

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

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

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

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue; // all observers in the shareability domain
                                                         // observe the load and store atomically

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

B11.1.4 CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory loads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in a first pair of registers. If the comparison is equal, the values in a second pair of registers are stored to memory.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	10	9	5	4	0
0	sz	0	0	1	0	0	0	0	L	1	Rs	o0	1	1	1	1	Rn	Rt	

Rt2

32-bit, acquire variant

Applies when $sz == 0 \ \&\& \ L == 1 \ \&\& \ o0 == 0$.

CASPA $\langle Ws \rangle$, $\langle W(s+1) \rangle$, $\langle Wt \rangle$, $\langle W(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

32-bit, acquire and release variant

Applies when $sz == 0 \ \&\& \ L == 1 \ \&\& \ o0 == 1$.

CASPAL $\langle Ws \rangle$, $\langle W(s+1) \rangle$, $\langle Wt \rangle$, $\langle W(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

32-bit, no memory ordering variant

Applies when $sz == 0 \ \&\& \ L == 0 \ \&\& \ o0 == 0$.

CASP $\langle Ws \rangle$, $\langle W(s+1) \rangle$, $\langle Wt \rangle$, $\langle W(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

32-bit, release variant

Applies when $sz == 0 \ \&\& \ L == 0 \ \&\& \ o0 == 1$.

CASPL $\langle Ws \rangle$, $\langle W(s+1) \rangle$, $\langle Wt \rangle$, $\langle W(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

64-bit, acquire variant

Applies when $sz == 1 \ \&\& \ L == 1 \ \&\& \ o0 == 0$.

CASPA $\langle Xs \rangle$, $\langle X(s+1) \rangle$, $\langle Xt \rangle$, $\langle X(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

64-bit, acquire and release variant

Applies when $sz == 1 \ \&\& \ L == 1 \ \&\& \ o0 == 1$.

CASPAL $\langle Xs \rangle$, $\langle X(s+1) \rangle$, $\langle Xt \rangle$, $\langle X(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

64-bit, no memory ordering variant

Applies when $sz == 1 \ \&\& \ L == 0 \ \&\& \ o0 == 0$.

CASP $\langle Xs \rangle$, $\langle X(s+1) \rangle$, $\langle Xt \rangle$, $\langle X(t+1) \rangle$, [$\langle Xn|SP \rangle\{, \#0\}$]

64-bit, release variant

Applies when `sz == 1 && L == 0 && o0 == 1`.

CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
if Rs<0> == '1' then UnallocatedEncoding();
if Rt<0> == '1' then UnallocatedEncoding();

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

integer datasize = 32 << UInt(sz);
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue     = if BigEndian() then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, (2 * datasize) DIV 8, ldacctype];
if data == comparevalue then
```

```
Mem[address, (2 * datasize) DIV 8, stacctype] = newvalue;
// all observers in the shareability domain
// observe the load and store atomically

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

RETIRED

B11.1.5 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDADD.
 - If the destination register is one of WZR or XZR, LDADDA and LDADDAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0	0	0	0	0	Rn	Rt		
size				V								o3			opc						

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDADDA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDADDAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDADD <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDADDL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDADDA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDADDAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDADD <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```

```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.6 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDADDB.
 - If the destination register is WZR, LDADDAB and LDADDALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	0	1	1	0	0	0	A	R	1		Rs	0	0	0	0	0	0		Rn		Rt
size				V								o3		opc							

Acquire variant

Applies when A == 1 && R == 0.

LDADDAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDADDALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDADDB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDADDLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;

```

```

when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.7 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDADDH.
 - If the destination register is WZR, LDADDAH and LDADDALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	0	0	0	0	0	Rn	Rt		
size				V								o3		opc							

Acquire variant

Applies when A == 1 && R == 0.

LDADDAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDADDALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDADDH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDADDLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;

```

```

when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.8 LDCLR, LDCLRA, LDCLRAL, LDCLRRL

Atomic bit clear on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRRL and LDCLRAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDCLR.
 - If the destination register is one of WZR or XZR, LDCLRA and LDCLRAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0	0	1	0	0	Rn	Rt		
size				V								o3			opc						

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDCLRA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDCLR <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDCLRRL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDCLRA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDCLRRL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDCLR <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDCLR <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```



```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.9 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

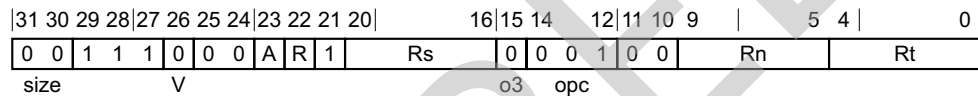
Atomic bit clear on byte in memory loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDCLRB.
 - If the destination register is WZR, LDCLRAB and LDCLRALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

LDCLRAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDCLRALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDCLRB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDCLRLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.10 LDCLR, LDCLRAH, LDCLRALH, LDCLRLH

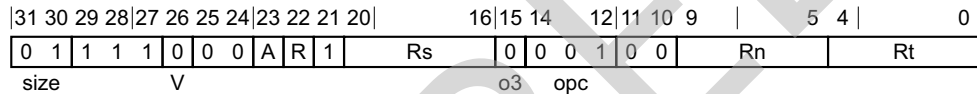
Atomic bit clear on halfword in memory loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDCLR.
 - If the destination register is WZR, LDCLRAH and LDCLRALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

LDCLRAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDCLRALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDCLR <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDCLRLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.11 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDEOR.
 - If the destination register is one of WZR or XZR, LDEORA and LDEORAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1		Rs	0	0	1	0	0	0	Rn		Rt
size				V									o3		opc						

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDEORA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDEORAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDEOR <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDEORL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDEORA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDEORAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDEOR <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDEORL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```

```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.12 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDEORB.
 - If the destination register is WZR, LDEORAB and LDEORALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31 30 29 28				27 26 25 24				23 22 21 20				16 15 14				12 11 10 9				5 4				0			
0 0		1 1 1		0 0 0		A R 1		Rs				0 0 1 0		0 0		Rn				Rt							
size				V								o3		opc													

Acquire variant

Applies when A == 1 && R == 0.

LDEORAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDEORALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDEORB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDEORLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.13 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDEORH.
 - If the destination register is WZR, LDEORAH and LDEORALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	0	1	0	0	0	Rn			Rt
size				V								o3			opc						

Acquire variant

Applies when A == 1 && R == 0.

LDEORAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDEORALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDEORH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDEORLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.14 LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B11-227. For information about memory accesses, see *Load/Store addressing modes* on page B11-228.

ARMv8.1

31 30 29 28 27 26 25 24 23 22 21 20								16 15 14						10 9		5 4		0			
1 x		0 0 1 0 0 0						1 1		0 (1) (1) (1) (1) (1) (1)						0 (1) (1) (1) (1) (1) (1)		Rn		Rt	
size								L				Rs						o0		Rt2	

32-bit variant

Applies when size == 10.

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

64-bit variant

Applies when size == 11.

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

Decode for all variants of this encoding

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

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler symbols

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

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

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

Operation

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

if n == 31 then
    CheckSPA1ignment();
    address = SP[];
else
    address = X[n];

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

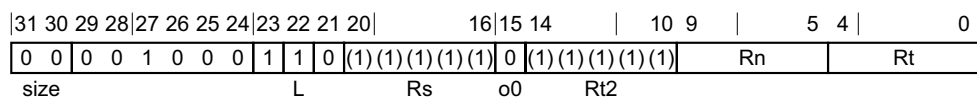
```
when MemOp_LOAD  
    data = Mem[address, dbytes, acctype];  
    X[t] = ZeroExtend(data, regsize);
```

RETIRED

B11.1.15 LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B11-227. For information about memory accesses, see *Load/Store addressing modes* on page B11-228.

ARMv8.1

**No offset variant**

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

Decode for this encoding

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

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler symbols

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

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

Operation

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

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

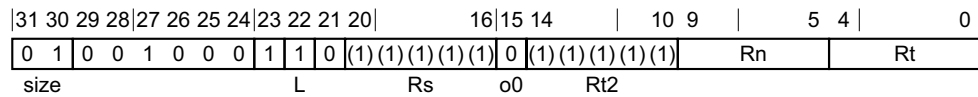
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

B11.1.16 LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#) on page B11-227. For information about memory accesses, see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No offset variant

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

Decode for this encoding

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

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler symbols

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

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

Operation

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

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

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

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


B11.1.17 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSET.
 - If the destination register is one of WZR or XZR, LDSETA and LDSETAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0	1	1	0	0	Rn	Rt		
size				V								o3			opc						

32-bit, acquire variant

Applies when size == 10 && A == 1 && R == 0.

LDSETA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when size == 10 && A == 1 && R == 1.

LDSETAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when size == 10 && A == 0 && R == 0 && Rt != 11111.

LDSET <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && A == 0 && R == 1 && Rt != 11111.

LDSETL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when size == 11 && A == 1 && R == 0.

LDSETA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when size == 11 && A == 1 && R == 1.

LDSETAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDSET <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDSETL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

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

value = X[s];
if n == 31 then
  CheckSPAAlignment();
  address = SP[];
else
  address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```

```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.18 LDSETB, LDSETAB, LDSETALB, LDSETLB

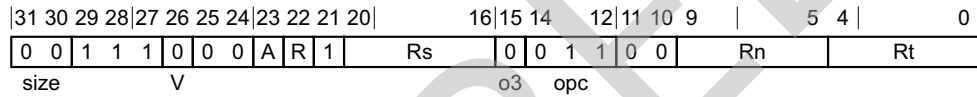
Atomic bit set on byte in memory loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSETB.
 - If the destination register is WZR, LDSETAB and LDSETALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

LDSETAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDSETALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDSETB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDSETLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.19 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSETH.
 - If the destination register is WZR, LDSETAH and LDSETALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	0	1	1	0	0	Rn			Rt
size				V								o3				opc					

Acquire variant

Applies when A == 1 && R == 0.

LDSETAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDSETALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDSETH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDSETLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.20 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSMAX.
 - If the destination register is one of WZR or XZR, LDSMAXA and LDSMAXAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1		Rs	0	1	0	0	0	0	Rn		Rt
size				V									o3		opc						

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDSMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDSMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDSMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDSMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDSMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDSMAXL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```

```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.21 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSMAXB.
 - If the destination register is WZR, LDSMAXAB and LDSMAXALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	0	0	0	0	Rn			Rt
size				V								o3			opc						

Acquire variant

Applies when A == 1 && R == 0.

LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.22 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSMAXH.
 - If the destination register is WZR, LDSMAXAH and LDSMAXALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1	0	0	0	0	Rn			Rt
size				V								o3				opc					

Acquire variant

Applies when A == 1 && R == 0.

LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDSMAXH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.23 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMINL and LDSMINAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSMIN.
 - If the destination register is one of WZR or XZR, LDSMINA and LDSMINAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1		Rs	0	1	0	1	0	0	Rn		Rt
size				V									o3		opc						

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDSMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDSMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDSMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDSMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDSMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDSMINL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```



```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.24 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

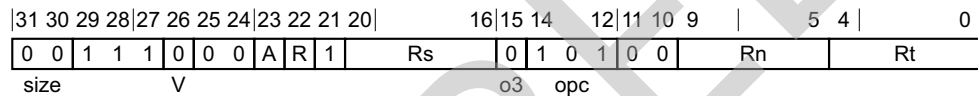
Atomic signed minimum on byte in memory loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSMINB.
 - If the destination register is WZR, LDSMINAB and LDSMINALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

LDSMINAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDSMINALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDSMINLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.25 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDSMINH.
 - If the destination register is WZR, LDSMINAH and LDSMINALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1	0	1	0	0	Rn			Rt
size				V								o3			opc						

Acquire variant

Applies when A == 1 && R == 0.

LDSMINAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDSMINALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDSMINH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDSMINLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.26 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDUMAX.
 - If the destination register is one of WZR or XZR, LDUMAXA and LDUMAXAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1	1	0	0	0	Rn			Rt
size		V										o3		opc							

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDUMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDUMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDUMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDUMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDUMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDUMAXL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```

```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.27 LDUMAXB, LDUMAXB, LDUMAXB, LDUMAXB

Atomic unsigned maximum on byte in memory loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXB and LDUMAXB load from memory with acquire semantics.
- LDUMAXB and LDUMAXB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDUMAXB.
 - If the destination register is WZR, LDUMAXB and LDUMAXB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	1	0	0	0	Rn			Rt
size				V								o3				opc					

Acquire variant

Applies when A == 1 && R == 0.

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
// observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.28 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDUMAXH.
 - If the destination register is WZR, LDUMAXAH and LDUMAXALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1	1	0	0	0	Rn			Rt
size				V								o3			opc						

Acquire variant

Applies when A == 1 && R == 0.

LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDUMAXH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.29 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDUMIN.
 - If the destination register is one of WZR or XZR, LDUMINA and LDUMINAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1		Rs	0	1	1	1	0	0	Rn		Rt
size				V									o3		opc						

32-bit, acquire variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDUMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when $\text{size} == 10 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 0 \ \&\& \ Rt \neq 11111$.

LDUMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when $\text{size} == 10 \ \&\& \ A == 0 \ \&\& \ R == 1 \ \&\& \ Rt \neq 11111$.

LDUMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 0$.

LDUMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when $\text{size} == 11 \ \&\& \ A == 1 \ \&\& \ R == 1$.

LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0 && Rt != 11111.

LDUMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1 && Rt != 11111.

LDUMINL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
```

```
case op of
  when MemAtomicOp_ADD    result = data + value;
  when MemAtomicOp_BIC    result = data AND NOT(value);
  when MemAtomicOp_EOR    result = data EOR value;
  when MemAtomicOp_ORR    result = data OR value;
  when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
  when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.30 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

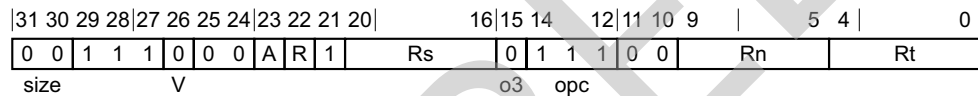
Atomic unsigned minimum on byte in memory loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDUMINB.
 - If the destination register is WZR, LDUMINAB and LDUMINALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

LDUMINAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDUMINALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDUMINB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDUMINLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```



```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
when MemAtomicOp_ADD result = data + value;
when MemAtomicOp_BIC result = data AND NOT(value);
when MemAtomicOp_EOR result = data EOR value;
when MemAtomicOp_ORR result = data OR value;
when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.31 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

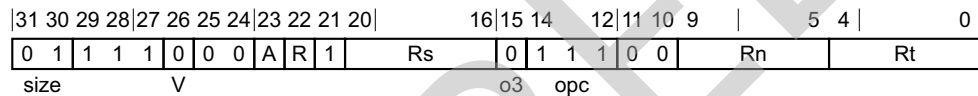
Atomic unsigned minimum on halfword in memory loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - LDUMINH.
 - If the destination register is WZR, LDUMINAH and LDUMINALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

LDUMINAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

LDUMINALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0 && Rt != 11111.

LDUMINH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1 && Rt != 11111.

LDUMINLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of

```

```

when '0000' op = MemAtomicOp_ADD;
when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.32 MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE, namely D, A, I, F, and SP. For more information, see "Process State, PSTATE" in the ARMv8-A Architecture Reference Manual.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	14	13	12	11	8	7	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	op1	0	1	0	0	CRm	op2	1	1	1	1	1	1	1

System variant

MSR <pstatefield>, #<imm>

Decode for this encoding

```
AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');

bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 100'
        if !HavePANExt() then
            UnallocatedEncoding();
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    otherwise      UnallocatedEncoding();

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == EL0 && (IsInHost() || SCTLRL_EL1.UMA == '0') then
    AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');
```

Assembler symbols

<pstatefield> Is a PSTATE field name, encoded in the "op1:op2" field. It can have the following values:

PAN	when op1 = 000, op2 = 100
SPSe1	when op1 = 000, op2 = 101
DAIFSet	when op1 = 011, op2 = 110
DAIFClr	when op1 = 011, op2 = 111

The following encodings are reserved:

- op1 = 000, op2 = 0xx.
- op1 = 000, op2 = 11x.
- op1 = 001, op2 = xxx.
- op1 = 010, op2 = xxx.
- op1 = 011, op2 = 0xx.
- op1 = 011, op2 = 10x.
- op1 = 1xx, op2 = xxx.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
case field of
  when PSTATEField_SP
    PSTATE.SP = operand<0>;
  when PSTATEField_DAIFSet
    PSTATE.D = PSTATE.D OR operand<3>;
    PSTATE.A = PSTATE.A OR operand<2>;
    PSTATE.I = PSTATE.I OR operand<1>;
    PSTATE.F = PSTATE.F OR operand<0>;
  when PSTATEField_DAIFClr
    PSTATE.D = PSTATE.D AND NOT(operand<3>);
    PSTATE.A = PSTATE.A AND NOT(operand<2>);
    PSTATE.I = PSTATE.I AND NOT(operand<1>);
    PSTATE.F = PSTATE.F AND NOT(operand<0>);
  when PSTATEField_PAN
    PSTATE.PAN = operand<0>;
```

RETIRED

B11.1.33 SQRDMLAH (by element)

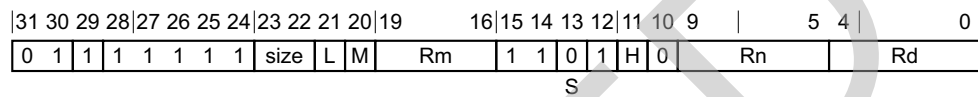
Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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

Scalar

ARMv8.1



Scalar variant

SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

```

if !HaveQRDMLAExt() then UnallocatedEncoding();

integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

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

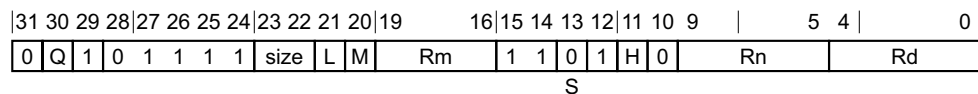
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

```

Vector

ARMv8.1



Vector variant

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

```

if !HaveQRDMLAExt() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

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

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

boolean rounding = TRUE;
boolean sub_op = (S == '1');

```

Assembler symbols

- <V> Is a width specifier, encoded in the "size" field. It can have the following values:
- | | |
|---|----------------|
| H | when size = 01 |
| S | when size = 10 |
- The following encodings are reserved:
- size = 00.
 - size = 11.
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- | | |
|----|-----------------------|
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |
- The following encodings are reserved:
- size = 00, Q = x.
 - size = 11, Q = x.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- | | |
|------|----------------|
| 0:Rm | when size = 01 |
| M:Rm | when size = 10 |
- The following encodings are reserved:
- size = 00.
 - size = 11.
- Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizesize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elm[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elm[operand1, e, esize]);
    element3 = SInt(Elm[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elm[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


B11.1.34 SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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

Scalar

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16 15 14 13 12 11 10 9								5 4		0	
0	1	1	1	1	1	1	0	size	0	Rm			1	0	0	0	0	1	Rn			Rd	
S																							

Scalar variant

SQRDMLAH <V><d>, <V><n>, <V><m>

Decode for this encoding

```

if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

```

Vector

ARMv8.1

31 30 29 28 27 26 25 24 23 22 21 20												16 15 14 13 12 11 10 9								5 4		0	
0	Q	1	0	1	1	1	0	size	0	Rm			1	0	0	0	0	1	Rn			Rd	
S																							

Vector variant

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```

if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

```

```
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler symbols

<V>	Is a width specifier, encoded in the "size" field. It can have the following values: H when size = 01 S when size = 10 The following encodings are reserved: • size = 00. • size = 11.
<d>	Is the number of the SIMD&FP destination register, in the "Rd" field.
<n>	Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m>	Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values: 4H when size = 01, Q = 0 8H when size = 01, Q = 1 2S when size = 10, Q = 0 4S when size = 10, Q = 1 The following encodings are reserved: • size = 00, Q = x. • size = 11, Q = x.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdySIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elm[operand1, e, esize]);
    element2 = SInt(Elm[operand2, e, esize]);
    element3 = SInt(Elm[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elm[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

B11.1.35 SQRDMLSH (by element)

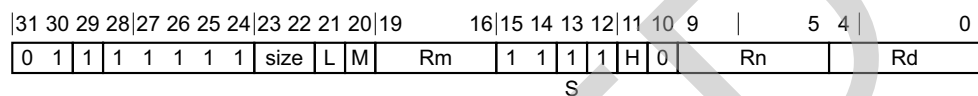
Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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

Scalar

ARMv8.1



Scalar variant

SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

```

if !HaveQRDMLAExt() then UnallocatedEncoding();

integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding();

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

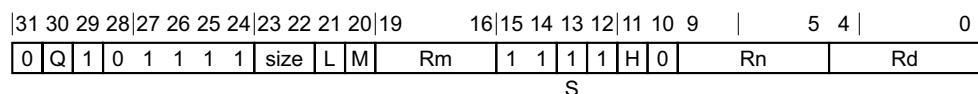
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

```

Vector

ARMv8.1



Vector variant

SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

```

if !HaveQRDMLAExt() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

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

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

boolean rounding = TRUE;
boolean sub_op = (S == '1');

```

Assembler symbols

- <V> Is a width specifier, encoded in the "size" field. It can have the following values:
- | | |
|---|----------------|
| H | when size = 01 |
| S | when size = 10 |
- The following encodings are reserved:
- size = 00.
 - size = 11.
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- | | |
|----|-----------------------|
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |
- The following encodings are reserved:
- size = 00, Q = x.
 - size = 11, Q = x.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- | | |
|------|----------------|
| 0:Rm | when size = 01 |
| M:Rm | when size = 10 |
- The following encodings are reserved:
- size = 00.
 - size = 11.
- Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01
H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizesize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elm[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elm[operand1, e, esize]);
    element3 = SInt(Elm[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elm[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

B11.1.36 SQRDMLSH (vector)

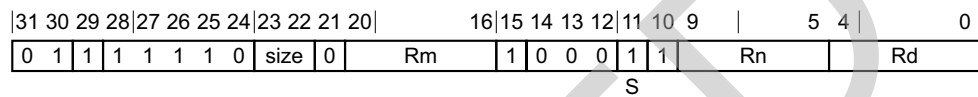
Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

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

Scalar

ARMv8.1



Scalar variant

SQRDMLSH <V><d>, <V><n>, <V><m>

Decode for this encoding

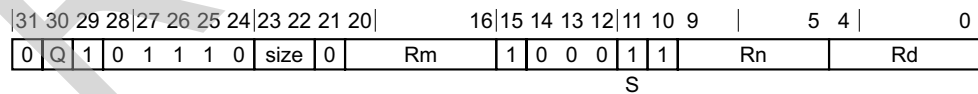
```

if !HaveQRDMLAExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

ARMv8.1



Vector variant

SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```

if !HaveQRDMLAExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler symbols

- <V> Is a width specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
 - S when size = 10
- The following encodings are reserved:
- size = 00.
 - size = 11.
- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 4H when size = 01, Q = 0
 - 8H when size = 01, Q = 1
 - 2S when size = 10, Q = 0
 - 4S when size = 10, Q = 1
- The following encodings are reserved:
- size = 00, Q = x.
 - size = 11, Q = x.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdySIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elm[operand1, e, esize]);
    element2 = SInt(Elm[operand2, e, esize]);
    element3 = SInt(Elm[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elm[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

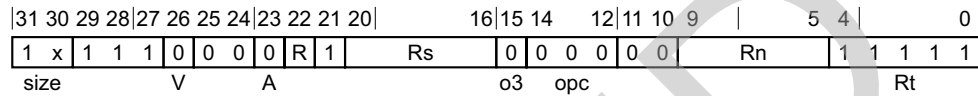
B11.1.37 STADD, STADDL

Atomic add on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD has no memory ordering semantics.
- STADDL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STADD <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STADDL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STADD <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STADDL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;

```



```
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

B11.1.38 STADDB, STADDLB

Atomic add on byte in memory, without return, loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB has no memory ordering semantics.
- STADDLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0			
0	0	1	1	1	0	0	0	0	R	1		Rs	0	0	0	0	0	0	Rn	1	1	1	1	1
size				V			A						o3		opc					Rt				

No memory ordering variant

Applies when R == 0.

STADDB <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STADDLB <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

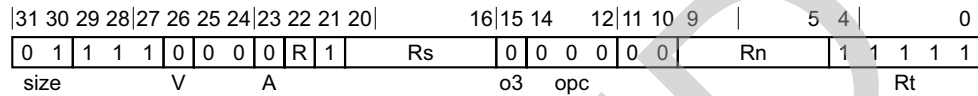
B11.1.39 STADDH, STADDLH

Atomic add on halfword in memory, without return, loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH has no memory ordering semantics.
- STADDLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STADDH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STADDLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

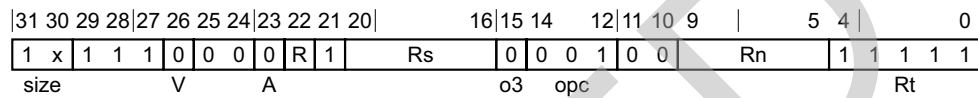
B11.1.40 STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR has no memory ordering semantics.
- STCLRL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STCLR <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STCLRL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STCLR <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STCLRL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;

```

```
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnalignedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

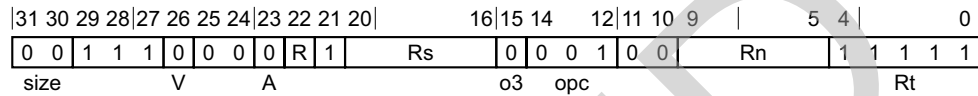
B11.1.41 STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB has no memory ordering semantics.
- STCLRLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



No memory ordering variant

Applies when R == 0.

STCLRB <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STCLRLB <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

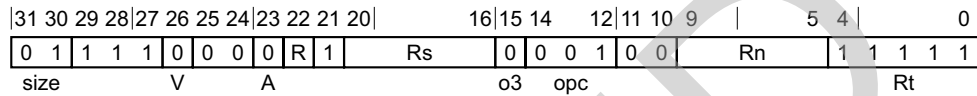
B11.1.42 STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH has no memory ordering semantics.
- STCLRLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



No memory ordering variant

Applies when R == 0.

STCLRH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STCLRLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.43 STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR has no memory ordering semantics.
- STEORL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0		
1	x	1	1	1	0	0	0	0	R	1		Rs	0	0	1	0	0	0	Rn	1	1	1	1
size				V				A				o3				opc				Rt			

32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STEOR <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STEORL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STEOR <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STEORL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;

```

```
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnalignedEncoding();
```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

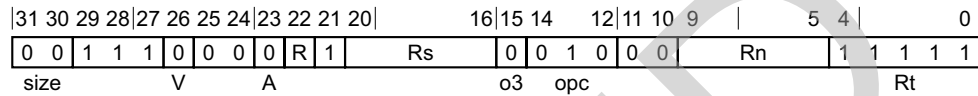
B11.1.44 STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB has no memory ordering semantics.
- STEORLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STEORB <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STEORLB <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

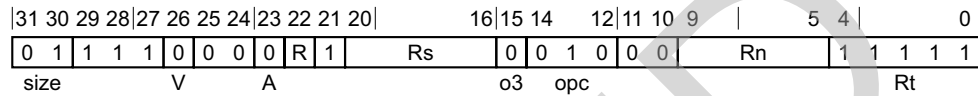
B11.1.45 STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH has no memory ordering semantics.
- STEORLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



No memory ordering variant

Applies when R == 0.

STEORH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STEORLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

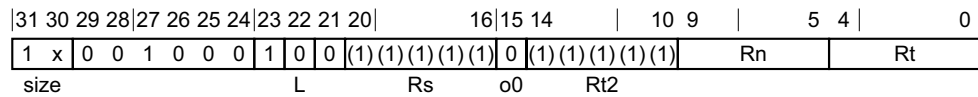
X[t] = ZeroExtend(data, regsize);

```

B11.1.46 STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B11-227. For information about memory accesses, see *Load/Store addressing modes* on page B11-228.

ARMv8.1



32-bit variant

Applies when size == 10.

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

64-bit variant

Applies when size == 11.

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

Decode for all variants of this encoding

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

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler symbols

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

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

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

Operation

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

if n == 31 then
    CheckSPA1ignment();
    address = SP[];
else
    address = X[n];

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

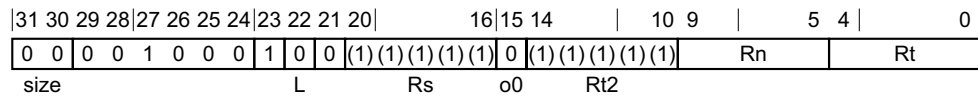
```
when MemOp_LOAD  
    data = Mem[address, dbytes, acctype];  
    X[t] = ZeroExtend(data, regsize);
```

RETIRED

B11.1.47 STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B11-227. For information about memory accesses, see *Load/Store addressing modes* on page B11-228.

ARMv8.1



No offset variant

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

Decode for this encoding

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

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler symbols

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

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

Operation

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

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

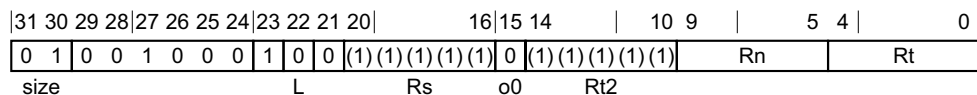
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

B11.1.48 STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B11-227. For information about memory accesses, see *Load/Store addressing modes* on page B11-228.

ARMv8.1

**No offset variant**

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

Decode for this encoding

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

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler symbols

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

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

Operation

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

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

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

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

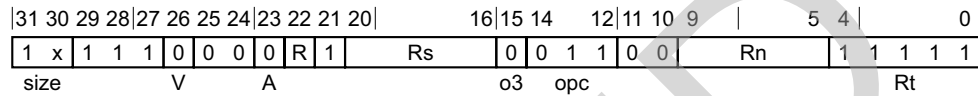
B11.1.49 STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET has no memory ordering semantics.
- STSETL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STSET <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STSETL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STSET <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STSETL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;

```

```
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

value = X[s];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

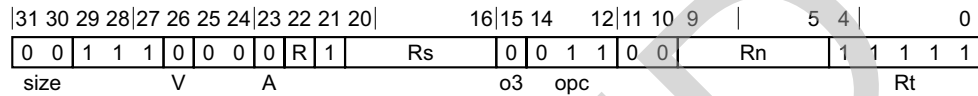
B11.1.50 STSETB, STSETLB

Atomic bit set on byte in memory, without return, loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB has no memory ordering semantics.
- STSETLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



No memory ordering variant

Applies when R == 0.

STSETB <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STSETLB <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

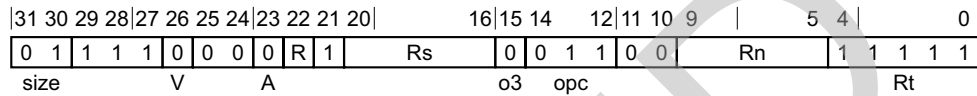
B11.1.51 STSETH, STSETLH

Atomic bit set on halfword in memory, without return, loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH has no memory ordering semantics.
- STSETLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STSETH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STSETLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

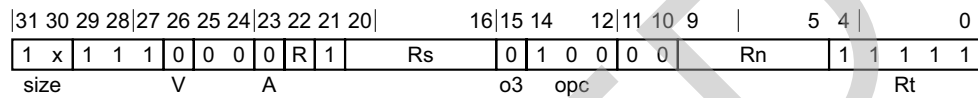
B11.1.52 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX has no memory ordering semantics.
- STSMAXL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STSMAX <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STSMAXL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STSMAX <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STSMAXL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;

```

```
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
// observe the load and store atomically

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

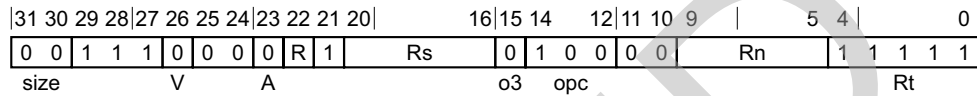
B11.1.53 STSMAXB, STSMAXB

Atomic signed maximum on byte in memory, without return, loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXB has no memory ordering semantics.
- STSMAXB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STSMAXB <Ws>, [<Xn>|SP>]

Release variant

Applies when R == 1.

STSMAXB <Ws>, [<Xn>|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn>|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

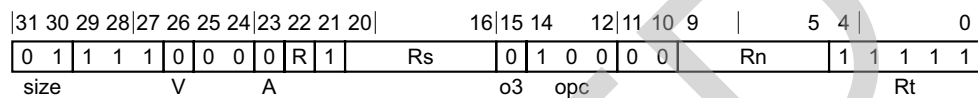
B11.1.54 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXH has no memory ordering semantics.
- STSMAXLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STSMAXH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STSMAXLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType staccctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

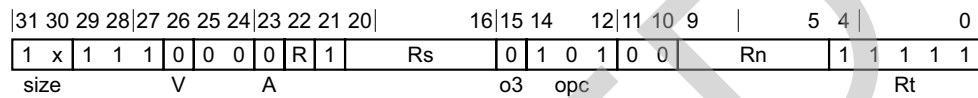
B11.1.55 STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN has no memory ordering semantics.
- STSMINL stores to memory with release semantics, as described in [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STSMIN <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STSMINL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STSMIN <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STSMINL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;

```

```
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnalignedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
// observe the load and store atomically

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

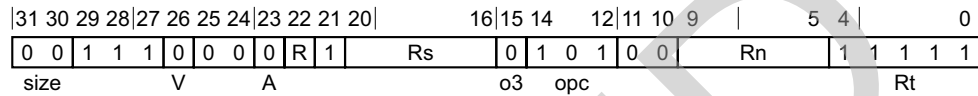
B11.1.56 STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINB has no memory ordering semantics.
- STSMINLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STSMINB <Ws>, [<Xn>|SP>]

Release variant

Applies when R == 1.

STSMINLB <Ws>, [<Xn>|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn>|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

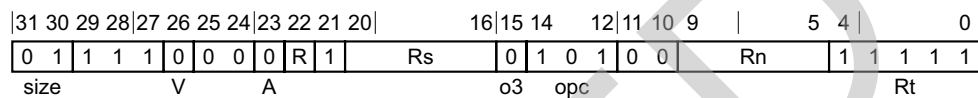
B11.1.57 STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH has no memory ordering semantics.
- STSMINLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STSMINH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STSMINLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType staccctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

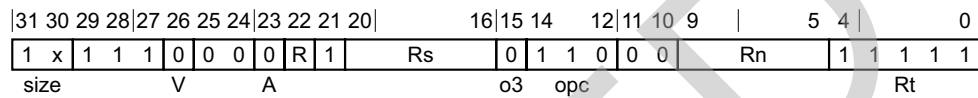
B11.1.58 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX has no memory ordering semantics.
- STUMAXL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STUMAX <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STUMAXL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STUMAX <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STUMAXL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;

```



```
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

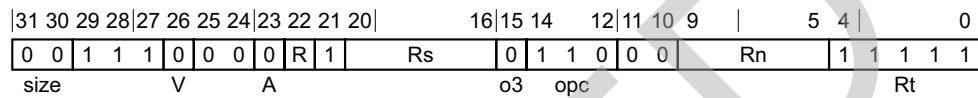
B11.1.59 STUMAXB, STUMAXB

Atomic unsigned maximum on byte in memory, without return, loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXB has no memory ordering semantics.
- STUMAXB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STUMAXB <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STUMAXB <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

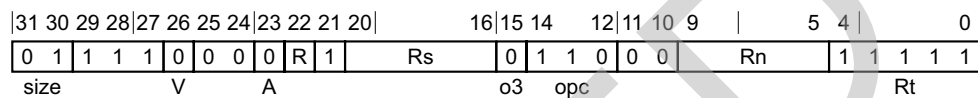
B11.1.60 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH has no memory ordering semantics.
- STUMAXLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STUMAXH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STUMAXLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

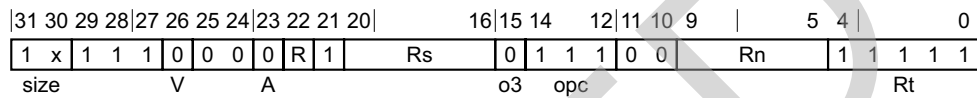
B11.1.61 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMIN has no memory ordering semantics.
- STUMINL stores to memory with release semantics, as described in [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



32-bit, no memory ordering variant

Applies when size == 10 && R == 0.

STUMIN <Ws>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && R == 1.

STUMINL <Ws>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && R == 0.

STUMIN <Xs>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && R == 1.

STUMINL <Xs>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;

```

```
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
// observe the load and store atomically

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

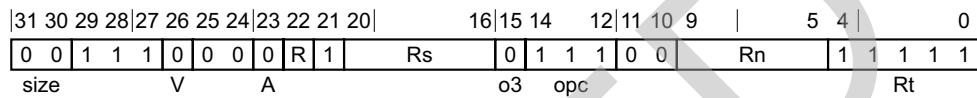
B11.1.62 STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINB has no memory ordering semantics.
- STUMINLB stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STUMINB <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STUMINLB <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType staccctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

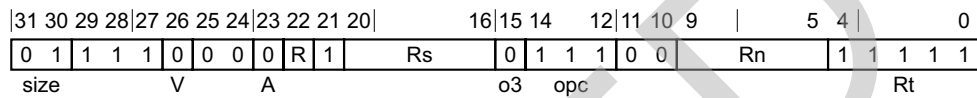
B11.1.63 STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH has no memory ordering semantics.
- STUMINLH stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1



No memory ordering variant

Applies when R == 0.

STUMINH <Ws>, [<Xn|SP>]

Release variant

Applies when R == 1.

STUMINLH <Ws>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType staccctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;
  when '0001' op = MemAtomicOp_BIC;
  when '0010' op = MemAtomicOp_EOR;
  when '0011' op = MemAtomicOp_ORR;
  when '0100' op = MemAtomicOp_SMAX;
  when '0101' op = MemAtomicOp_SMIN;
  when '0110' op = MemAtomicOp_UMAX;
  when '0111' op = MemAtomicOp_UMIN;
  when '1000' op = MemAtomicOp_SWP;
  otherwise UnallocatedEncoding();

```

Assembler symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.1.64 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- The following have no memory ordering requirements:
 - SWP.
 - If the destination register is one of WZR or XZR, SWPA and SWPAL.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#) on page B11-227.

For information about memory accesses see [Load/Store addressing modes](#) on page B11-228.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	16	15	14	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	A	R	1	Rs	1	0	0	0	0	0	Rn			Rt
size				V								o3				opc					

32-bit, acquire variant

Applies when size == 10 && A == 1 && R == 0.

SWPA <Ws>, <Wt>, [<Xn|SP>]

32-bit, acquire and release variant

Applies when size == 10 && A == 1 && R == 1.

SWPAL <Ws>, <Wt>, [<Xn|SP>]

32-bit, no memory ordering variant

Applies when size == 10 && A == 0 && R == 0.

SWP <Ws>, <Wt>, [<Xn|SP>]

32-bit, release variant

Applies when size == 10 && A == 0 && R == 1.

SWPL <Ws>, <Wt>, [<Xn|SP>]

64-bit, acquire variant

Applies when size == 11 && A == 1 && R == 0.

SWPA <Xs>, <Xt>, [<Xn|SP>]

64-bit, acquire and release variant

Applies when size == 11 && A == 1 && R == 1.

SWPAL <Xs>, <Xt>, [<Xn|SP>]

64-bit, no memory ordering variant

Applies when size == 11 && A == 0 && R == 0.

SWP <Xs>, <Xt>, [<Xn|SP>]

64-bit, release variant

Applies when size == 11 && A == 0 && R == 1.

SWPL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
    when '0000' op = MemAtomicOp_ADD;
    when '0001' op = MemAtomicOp_BIC;
    when '0010' op = MemAtomicOp_EOR;
    when '0011' op = MemAtomicOp_ORR;
    when '0100' op = MemAtomicOp_SMAX;
    when '0101' op = MemAtomicOp_SMIN;
    when '0110' op = MemAtomicOp_UMAX;
    when '0111' op = MemAtomicOp_UMIN;
    when '1000' op = MemAtomicOp_SWP;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

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

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

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

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

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
```

```
when MemAtomicOp_BIC    result = data AND NOT(value);
when MemAtomicOp_EOR    result = data EOR value;
when MemAtomicOp_ORR    result = data OR value;
when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

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

RETIRED

B11.1.65 SWPB, SWPAB, SWPALB, SWPLB

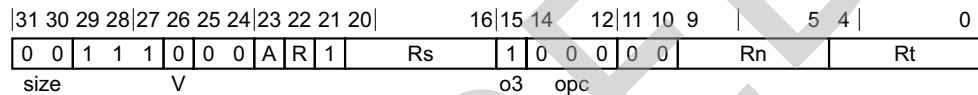
Swap byte in memory loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- The following have no memory ordering requirements:
 - SWPB.
 - If the destination register is WZR, SWPAB and SWPALB.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



Acquire variant

Applies when A == 1 && R == 0.

SWPAB <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when A == 1 && R == 1.

SWPALB <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when A == 0 && R == 0.

SWPB <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when A == 0 && R == 1.

SWPLB <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;

```

```

when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP    result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```


B11.1.66 SWPH, SWPAH, SWPALH, SWPLH

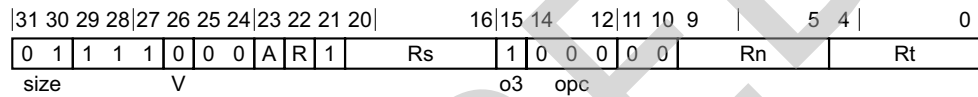
Swap halfword in memory loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- The following have no memory ordering requirements:
 - SWPH.
 - If the destination register is WZR, SWPAH and SWPALH.

For more information about memory ordering semantics see [Load-Acquire, Store-Release on page B11-227](#).

For information about memory accesses see [Load/Store addressing modes on page B11-228](#).

ARMv8.1



Acquire variant

Applies when $A == 1 \ \&\& \ R == 0$.

SWPAH <Ws>, <Wt>, [<Xn|SP>]

Acquire and release variant

Applies when $A == 1 \ \&\& \ R == 1$.

SWPALH <Ws>, <Wt>, [<Xn|SP>]

No memory ordering variant

Applies when $A == 0 \ \&\& \ R == 0$.

SWPH <Ws>, <Wt>, [<Xn|SP>]

Release variant

Applies when $A == 0 \ \&\& \ R == 1$.

SWPLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

```

if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case o3:opc of
  when '0000' op = MemAtomicOp_ADD;

```

```

when '0001' op = MemAtomicOp_BIC;
when '0010' op = MemAtomicOp_EOR;
when '0011' op = MemAtomicOp_ORR;
when '0100' op = MemAtomicOp_SMAX;
when '0101' op = MemAtomicOp_SMIN;
when '0110' op = MemAtomicOp_UMAX;
when '0111' op = MemAtomicOp_UMIN;
when '1000' op = MemAtomicOp_SWP;
otherwise UnallocatedEncoding();

```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

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

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

Operation

```

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
    when MemAtomicOp_SWP result = value;

Mem[address, datasize DIV 8, stacctype] = result; // all observers in the shareability domain
                                                    // observe the load and store atomically

X[t] = ZeroExtend(data, regsize);

```

B11.2 ARMv8.0 sections relating to these instructions

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* are included in this supplement to complement the instruction descriptions.

B11.2.1 Load-Acquire, Store-Release

ARMv8 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores.

For all memory types, these instructions have the following ordering requirements:

- A Store-Release followed by a Load-Acquire is observed in program order by any observers that are in both:
 - The shareability domain of the address accessed by the Store-Release.
 - The shareability domain of the address accessed by the Load-Acquire.
- For a Load-Acquire, observers in the shareability domain of the address accessed by the Load-Acquire observe accesses in the following order:
 1. The read caused by the Load-Acquire.
 2. Reads and writes caused by loads and stores that appear in program order after the Load-Acquire for which the shareability of the address accessed by the load or store requires that the observer observes the access.

There are no additional ordering requirements on loads or stores that appear before the Load-Acquire.

- For a Store-Release, observers in the shareability domain of the address accessed by the Store-Release observe accesses in the following order:
 1. All of the following for which the shareability of the address accessed requires that the observer observes the access:
 - Reads and writes caused by loads and stores that appear in program order before the Store-Release.
 - Writes that were observed by the PE executing the Store-Release before it executed the Store-Release.
 2. The write caused by the Store-Release.

There are no other ordering requirements on loads or stores that appear in program order after the Store-Release.

- A Store-Release instruction is multi-copy atomic when observed with a Load-Acquire instruction.

In addition, for accesses to a memory-mapped peripheral of an arbitrary system-defined size that are defined as any type of Device memory accesses, these instructions have the following requirements:

- A Load-Acquire to an address in the memory-mapped peripheral will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load-Acquire will arrive at the memory-mapped peripheral after the memory access of the Load-Acquire.
- A Store-Release to an address in the memory-mapped peripheral will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed before the Store-Release will arrive at the memory-mapped peripheral before the memory access of the Store-Release.
- If a Load-Acquire to a memory address in the memory-mapped peripheral has observed the value stored to that address by a Store-Release, then any memory access to the memory-mapped peripheral that is architecturally required to be ordered before the memory access of the Store-Release will arrive at the memory-mapped peripheral before any memory access to the same peripheral that is architecturally required to be ordered after the memory access of the Load-Acquire.

Load-Acquire and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements. The address supplied to the instructions must be aligned to twice the size of the element being loaded, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the release semantics if the store is successful.

———— **Note** ————

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
 - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
 - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction.

B11.2.2 Load/Store addressing modes

Load/Store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, SP, with an optional immediate or register offset. [Table B11-1](#) shows the assembler syntax for the complete set of Load/Store addressing modes.

Table B11-1 A64 Load/Store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	[base{, #0}]	-	-
Base plus offset	[base{, #imm}]	[base, Xm{, LSL #imm}]	[base, Wm, (S U)XTW {#imm}]
Pre-indexed	[base, #imm]!	-	-
Post-indexed	[base], #imm	[base], Xm ^a	-
Literal (PC-relative)	label	-	-

- a. The post-indexed by register offset mode can be used with the SIMD Load/Store structure instructions described in the section *Load/Store Vector in Chapter C5 of the ARM ARM*. Otherwise the post-indexed by register offset mode is not available.

Some types of Load/Store instruction support only a subset of the Load/Store addressing modes listed in [Table B11-1](#). Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within $\pm 1\text{MB}$ of the address of this instruction with no offset. Literal addressing can only be used for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.

- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of Load/Store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of Load/Store instruction and the transfer size.

Table B11-2 shows the offset and the type of Load/Store instruction.

Table B11-2 Immediate offsets and the type of Load/Store instruction

Offset bits	Sign	Scaling	Write-Back	Load/Store type
0	-	-	-	Exclusive/acquire/release
7	Signed	Scaled	Optional	Register pair
9	Signed	Unscaled	Optional	Single register
12	Unsigned	Scaled	No	Single register

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if LSL #imm is present and where imm must be equal to $\log_2(\text{transfer_size})$.
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by #imm, where imm must be equal to $\log_2(\text{transfer_size})$. An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific Load/Store instruction requires this. SP cannot be used as a register offset.

Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Table B11-3 shows the arithmetic instructions that can compute addressing modes.

Table B11-3 Arithmetic instructions to compute addressing modes

Addressing Form	Offset		
	Immediate	Register	Extended Register
Base register (no offset)	MOV Xd SP, base	-	-
Base plus offset	ADD Xd SP, base, #imm or SUB Xd SP, base, #imm	ADD <Xd SP>, base, Xm{,LSL#imm}	ADD <Xd SP>, base, Wm,(S U)XT(W H B) {#imm}
Pre-indexed	-	-	-
Post-indexed	-	-	-
Literal (PC-relative)	ADR Xd, label	-	-

Note

- To calculate a base plus immediate offset, the ADD instructions accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single ADD instruction cannot support the full range of byte offsets available to a single register Load/Store with a scaled 12-bit immediate offset. For example, a quadword LDR effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two ADD instructions. The following example shows this:

```
ADD Xd, base, #(imm & 0xFFF)
ADD Xd, Xd, #(imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the ADD instructions provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```
ADD Xd, base, Wm, SXTW #3    // Xd = base + (SignExtend(Wm) LSL 3)
ADD Xd, base, Wm, UXTH #4    // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one Load/Store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then re-use it as a simple register offset. The extend and scale calculation can be performed using the SBFIZ and UBFIZ bitfield instructions defined in the section *Bitfield move in Chapter C3 of the ARM ARM*, for example:

```
SBFIZ Xd, Xm, #3, #32    //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16    //Xd = "Wm, UXTH #4"
```

Chapter B12

AArch64 Register Descriptions

This chapter describes the AArch64 System registers that are added or affected by ARMv8.1. It contains the following sections:

- *General information about AArch64 System registers on page B12-232.*
- *General system control registers on page B12-234.*
- *Debug registers on page B12-424.*
- *Performance Monitors registers on page B12-450*
- *Generic Timer registers on page B12-463*
- *System instructions on page B12-504*
- *ARMv8.0 sections relating to these registers on page B12-545.*

B12.1 General information about AArch64 System registers

The structure of the System register descriptions has changed from that used in *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, Issue A.j and earlier:

- Information about the accessibility of the register from different Exception levels is given in the Accessibility section, that is the last but one section of a register description.
- Information about the traps and enables that apply to the register is given in the Traps and Enables section, that is the last section of a register description.

The information in these sections can depend on:

- The mnemonic that is used to access the register.
- The value of one or more of the controls {E2H, TGE, NS}.

These controls are:

- [HCR_EL2](#).{E2H, TGE} fields.
- [SCR_EL3](#).NS field.

Note

- These changes mean the registers descriptions can address:
 - Cases where a single register is accessible using more than one mnemonic, in different contexts, and that the accessibility can depend on the mnemonic used and the context in which it is used.
 - Cases where a single mnemonic can address different registers, depending on the context, and that the accessibility can also depend on the context.

These changes are needed to describe System register behaviors associated with the Virtualization Host Extension described in [Chapter B8 Virtualization Host Extensions](#). However, they also improve the representation of many ARMv8.0 register descriptions.

- This change to the structure of System register descriptions does not apply to the description of memory-mapped registers such as those described in [Chapter D2 External Debug Register Descriptions](#).

B12.1.1 The register descriptions included in this supplement

In general, this supplement includes the full description of all registers that are changed by ARMv8.1, including registers where the only changes introduced by ARMv8.1 are to the accessibility of the register. The only exception to this is the [ESR_ELx](#) register description, see [Changes to ESR_ELx](#).

The AArch64 System registers descriptions in this chapter do not highlight where ARMv8.1 has changed the register field descriptions. However:

- The field descriptions indicate any differences in behavior between ARMv8.0 and ARMv8.1.
- The descriptions of the features of ARMv8.1 elsewhere in this manual indicate where ARMv8.1 has introduced new register fields, or significantly changed the effect of a register field.

In addition, for the following register descriptions, the effect of [HCR_EL2](#).{E2H, TGE} on the register syntax is such that separate syntax descriptions are provided for different values of one or both of these fields:

- [SCTLR_EL2](#), *System Control Register (EL2)* on page B12-344.
- [TCR_EL2](#), *Translation Control Register (EL2)* on page B12-394.
- [CNTHCTL_EL2](#), *Counter-timer Hypervisor Control register* on page B12-464.

B12.1.2 Changes to ESR_ELx

ARMv8.1 architecture makes only limited changes to the [ESR_ELx](#) field descriptions, and this register is not reproduced in this supplement. However, the register descriptions include [ESR_EL1](#), [ESR_EL2](#), and [ESR_EL3](#). For the field descriptions of these registers, see the ARMv8 ARM.

———— **Note** ————

The ESR_ELx register is not included in this supplement because its description is very extensive, with many cross-references to other sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, meaning it has to be read in the context of that document.

The ESR_ELx changes introduced in the ARMv8.1 architecture are:

- If atomic instructions are not atomic in regard to other agents that access memory, then performing an atomic instruction to such a location generates an IMPLEMENTATION DEFINED MMU fault reported using the new Fault Status code of ESR_ELx.DFSC = 110101 for Data Aborts.
- If hardware updates of the translation tables are not atomic in regard to other agents that access memory, then performing a hardware update to such a location generates an Unsupported atomic hardware update MMU fault reported using the new Fault Status code of:
 - ESR_ELx.DFSC = 110001 for Data Aborts.
 - ESR_ELx.IFSC = 110001 for Instruction Aborts.

For the Non-secure EL1&0 translation regime, if atomic instruction or atomic hardware update is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.

B12.2 General system control registers

This section lists the ARMv8.1 System registers in AArch64 state that are not part of one of the other listed groups.

RETIRED

B12.2.1 ACTLR_EL1, Auxiliary Control Register (EL1)

The ACTLR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

Note

ARM recommends the contents of this register have no effect on the PE when [HCR_EL2](#).{E2H, TGE} is {1, 1}, and instead the configuration and control fields are provided by the [ACTLR_EL2](#) register. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

Configurations

AArch64 System register ACTLR_EL1[31:0] is architecturally mapped to AArch32 System register ACTLR.

AArch64 System register ACTLR_EL1[63:32] is architecturally mapped to AArch32 System register ACTLR2.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

ACTLR_EL1 is a 64-bit register.

Field descriptions

The ACTLR_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the ACTLR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ACTLR_EL1	11	000	0001	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ACTLR_EL1	x	x	0	-	RW	n/a	RW
ACTLR_EL1	0	0	1	-	RW	RW	RW
ACTLR_EL1	0	1	1	-	n/a	RW	RW
ACTLR_EL1	1	0	1	-	RW	RW	RW
ACTLR_EL1	1	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TACR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TACR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

B12.2.2 ACTLR_EL2, Auxiliary Control Register (EL2)

The ACTLR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for EL2.

Note

ARM recommends the contents of this register are updated to apply to EL0 when [HCR_EL2](#).{E2H, TGE} is {1, 1}, gaining configuration and control fields from the [ACTLR_EL1](#). This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

Configurations

AArch64 System register ACTLR_EL2[31:0] is architecturally mapped to AArch32 System register HACTLR.

AArch64 System register ACTLR_EL2[63:32] is architecturally mapped to AArch32 System register HACTLR2.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

ACTLR_EL2 is a 64-bit register.

Field descriptions

The ACTLR_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the ACTLR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ACTLR_EL2	11	100	0001	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ACTLR_EL2	x	x	0	-	-	n/a	RW
ACTLR_EL2	0	0	1	-	-	RW	RW
ACTLR_EL2	0	1	1	-	n/a	RW	RW
ACTLR_EL2	1	0	1	-	-	RW	RW
ACTLR_EL2	1	1	1	-	n/a	RW	RW

B12.2.3 AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)

The AFSR0_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

Configurations

AArch64 System register AFSR0_EL1 is architecturally mapped to AArch32 System register AFSR.

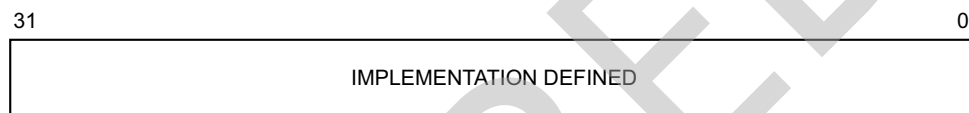
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

AFSR0_EL1 is a 32-bit register.

Field descriptions

The AFSR0_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the AFSR0_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
AFSR0_EL1	11	000	0101	0001	000
AFSR0_EL12	11	101	0101	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
AFSR0_EL1	x	x	0	-	RW	n/a	RW
AFSR0_EL1	0	0	1	-	RW	RW	RW
AFSR0_EL1	0	1	1	-	n/a	RW	RW
AFSR0_EL1	1	0	1	-	RW	AFSR0_EL2	RW
AFSR0_EL1	1	1	1	-	n/a	AFSR0_EL2	RW
AFSR0_EL12	x	x	0	-	-	n/a	-
AFSR0_EL12	0	0	1	-	-	-	-
AFSR0_EL12	0	1	1	-	n/a	-	-
AFSR0_EL12	1	0	1	-	-	RW	RW
AFSR0_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic AFSR0_EL1 or AFSR0_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.4 AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)

The AFSR0_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

Configurations

AArch64 System register AFSR0_EL2 is architecturally mapped to AArch32 System register HADFSR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

AFSR0_EL2 is a 32-bit register.

Field descriptions

The AFSR0_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the AFSR0_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
AFSR0_EL2	11	100	0101	0001	000
AFSR0_EL1	11	000	0101	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
AFSR0_EL2	x	x	0	-	-	n/a	RW
AFSR0_EL2	0	0	1	-	-	RW	RW
AFSR0_EL2	0	1	1	-	n/a	RW	RW
AFSR0_EL2	1	0	1	-	-	RW	RW
AFSR0_EL2	1	1	1	-	n/a	RW	RW
AFSR0_EL1	x	x	0	-	AFSR0_EL1	n/a	AFSR0_EL1
AFSR0_EL1	0	0	1	-	AFSR0_EL1	AFSR0_EL1	AFSR0_EL1
AFSR0_EL1	0	1	1	-	n/a	AFSR0_EL1	AFSR0_EL1
AFSR0_EL1	1	0	1	-	AFSR0_EL1	RW	AFSR0_EL1
AFSR0_EL1	1	1	1	-	n/a	RW	AFSR0_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic AFSR0_EL2 or AFSR0_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.5 AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)

The AFSR1_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

Configurations

AArch64 System register AFSR1_EL1 is architecturally mapped to AArch32 System register AIFSR.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

AFSR1_EL1 is a 32-bit register.

Field descriptions

The AFSR1_EL1 bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the AFSR1_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
AFSR1_EL1	11	000	0101	0001	001
AFSR1_EL12	11	101	0101	0001	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
AFSR1_EL1	x	x	0	-	RW	n/a	RW
AFSR1_EL1	0	0	1	-	RW	RW	RW
AFSR1_EL1	0	1	1	-	n/a	RW	RW
AFSR1_EL1	1	0	1	-	RW	AFSR1_EL2	RW
AFSR1_EL1	1	1	1	-	n/a	AFSR1_EL2	RW
AFSR1_EL12	x	x	0	-	-	n/a	-
AFSR1_EL12	0	0	1	-	-	-	-
AFSR1_EL12	0	1	1	-	n/a	-	-
AFSR1_EL12	1	0	1	-	-	RW	RW
AFSR1_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic AFSR1_EL1 or AFSR1_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.6 AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)

The AFSR1_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

Configurations

AArch64 System register AFSR1_EL2 is architecturally mapped to AArch32 System register HAIFSR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

AFSR1_EL2 is a 32-bit register.

Field descriptions

The AFSR1_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the AFSR1_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
AFSR1_EL2	11	100	0101	0001	001
AFSR1_EL1	11	000	0101	0001	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
AFSR1_EL2	x	x	0	-	-	n/a	RW
AFSR1_EL2	0	0	1	-	-	RW	RW
AFSR1_EL2	0	1	1	-	n/a	RW	RW
AFSR1_EL2	1	0	1	-	-	RW	RW
AFSR1_EL2	1	1	1	-	n/a	RW	RW
AFSR1_EL1	x	x	0	-	AFSR1_EL1	n/a	AFSR1_EL1
AFSR1_EL1	0	0	1	-	AFSR1_EL1	AFSR1_EL1	AFSR1_EL1
AFSR1_EL1	0	1	1	-	n/a	AFSR1_EL1	AFSR1_EL1
AFSR1_EL1	1	0	1	-	AFSR1_EL1	RW	AFSR1_EL1
AFSR1_EL1	1	1	1	-	n/a	RW	AFSR1_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic AFSR1_EL2 or AFSR1_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.7 AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by [MAIR_EL1](#).

Configurations

AArch64 System register AMAIR_EL1[31:0] is architecturally mapped to AArch32 System register AMAIR0.

AArch64 System register AMAIR_EL1[63:32] is architecturally mapped to AArch32 System register AMAIR1.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

AMAIR_EL1 is a 64-bit register.

Field descriptions

The AMAIR_EL1 bit assignments are:



AMAIR_EL1 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the AMAIR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
AMAIR_EL1	11	000	1010	0011	000
AMAIR_EL12	11	101	1010	0011	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
AMAIR_EL1	x	x	0	-	RW	n/a	RW
AMAIR_EL1	0	0	1	-	RW	RW	RW
AMAIR_EL1	0	1	1	-	n/a	RW	RW
AMAIR_EL1	1	0	1	-	RW	AMAIR_EL2	RW
AMAIR_EL1	1	1	1	-	n/a	AMAIR_EL2	RW
AMAIR_EL12	x	x	0	-	-	n/a	-
AMAIR_EL12	0	0	1	-	-	-	-
AMAIR_EL12	0	1	1	-	n/a	-	-
AMAIR_EL12	1	0	1	-	-	RW	RW
AMAIR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic AMAIR_EL1 or AMAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.8 AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by [MAIR_EL2](#).

Configurations

AArch64 System register AMAIR_EL2[31:0] is architecturally mapped to AArch32 System register HMAIR0.

AArch64 System register AMAIR_EL2[63:32] is architecturally mapped to AArch32 System register HMAIR1.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

AMAIR_EL2 is a 64-bit register.

Field descriptions

The AMAIR_EL2 bit assignments are:



AMAIR_EL2 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing the AMAIR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
AMAIR_EL2	11	100	1010	0011	000
AMAIR_EL1	11	000	1010	0011	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
AMAIR_EL2	x	x	0	-	-	n/a	RW
AMAIR_EL2	0	0	1	-	-	RW	RW
AMAIR_EL2	0	1	1	-	n/a	RW	RW
AMAIR_EL2	1	0	1	-	-	RW	RW
AMAIR_EL2	1	1	1	-	n/a	RW	RW
AMAIR_EL1	x	x	0	-	AMAIR_EL1	n/a	AMAIR_EL1
AMAIR_EL1	0	0	1	-	AMAIR_EL1	AMAIR_EL1	AMAIR_EL1
AMAIR_EL1	0	1	1	-	n/a	AMAIR_EL1	AMAIR_EL1
AMAIR_EL1	1	0	1	-	AMAIR_EL1	RW	AMAIR_EL1
AMAIR_EL1	1	1	1	-	n/a	RW	AMAIR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic `AMAIR_EL2` or `AMAIR_EL1` are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.9 CONTEXTIDR_EL1, Context ID Register (EL1)

The CONTEXTIDR_EL1 characteristics are:

Purpose

Identifies the current Process Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

This register is used:

- In ARMv8.0.
- In ARMv8.1, when [HCR_EL2.E2H](#) is 0.

Note

In ARMv8.1, when [HCR_EL2.E2H](#) is set to 1, [CONTEXTIDR_EL2](#) is used.

Configurations

AArch64 System register CONTEXTIDR_EL1 is architecturally mapped to AArch32 System register CONTEXTIDR.

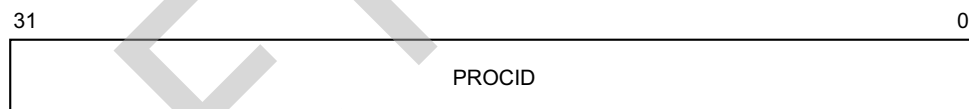
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CONTEXTIDR_EL1 is a 32-bit register.

Field descriptions

The CONTEXTIDR_EL1 bit assignments are:



PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

Note

In AArch32 state, when TTBCR.EAE is set to 0, CONTEXTIDR.ASID holds the ASID.

In AArch64 state, CONTEXTIDR_EL1 is independent of the ASID, and for the EL1&0 translation regime either [TTBR0_EL1](#) or [TTBR1_EL1](#) holds the ASID.

Accessing the CONTEXTIDR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CONTEXTIDR_EL1	11	000	1101	0000	001
CONTEXTIDR_EL12	11	101	1101	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CONTEXTIDR_EL1	x	x	0	-	RW	n/a	RW
CONTEXTIDR_EL1	0	0	1	-	RW	RW	RW
CONTEXTIDR_EL1	0	1	1	-	n/a	RW	RW
CONTEXTIDR_EL1	1	0	1	-	RW	CONTEXTIDR_EL2	RW
CONTEXTIDR_EL1	1	1	1	-	n/a	CONTEXTIDR_EL2	RW
CONTEXTIDR_EL12	x	x	0	-	-	n/a	-
CONTEXTIDR_EL12	0	0	1	-	-	-	-
CONTEXTIDR_EL12	0	1	1	-	n/a	-	-
CONTEXTIDR_EL12	1	0	1	-	-	RW	RW
CONTEXTIDR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic [CONTEXTIDR_EL1](#) or [CONTEXTIDR_EL12](#) are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.10 CONTEXTIDR_EL2, Context ID Register (EL2)

The CONTEXTIDR_EL2 characteristics are:

Purpose

In ARMv8.1, when `HCR_EL2.E2H` is set to 1, identifies the current Process Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

- Note

In ARMv8.0, and in ARMv8.1 when `HCR_EL2.E2H` is 0, `CONTEXTIDR_EL1` is used.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CONTEXTIDR_EL2 is a 32-bit register.

Field descriptions

The CONTEXTIDR_EL2 bit assignments are:

**PROCID, bits [31:0]**

Process Identifier. This field must be programmed with a unique value that identifies the current process.

- Note

In AArch32 state, when TTBCR.EAE is set to 0, CONTEXTIDR.ASID holds the ASID.

In AArch64 state, CONTEXTIDR_EL2 is independent of the ASID, and for the EL2&0 translation regime either **TTBR0_EL2** or **TTBR1_EL2** holds the ASID.

Accessing the CONTEXTIDR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CONTEXTIDR_EL2	11	100	1101	0000	001
CONTEXTIDR_EL1	11	000	1101	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CONTEXTIDR_EL2	x	x	0	-	-	n/a	RW
CONTEXTIDR_EL2	0	0	1	-	-	RW	RW
CONTEXTIDR_EL2	0	1	1	-	n/a	RW	RW
CONTEXTIDR_EL2	1	0	1	-	-	RW	RW
CONTEXTIDR_EL2	1	1	1	-	n/a	RW	RW
CONTEXTIDR_EL1	x	x	0	-	CONTEXTIDR_EL1	n/a	CONTEXTIDR_EL1
CONTEXTIDR_EL1	0	0	1	-	CONTEXTIDR_EL1	CONTEXTIDR_EL1	CONTEXTIDR_EL1
CONTEXTIDR_EL1	0	1	1	-	n/a	CONTEXTIDR_EL1	CONTEXTIDR_EL1
CONTEXTIDR_EL1	1	0	1	-	CONTEXTIDR_EL1	RW	CONTEXTIDR_EL1
CONTEXTIDR_EL1	1	1	1	-	n/a	RW	CONTEXTIDR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic `CONTEXTIDR_EL2` or `CONTEXTIDR_EL1` are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.11 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, and to Advanced SIMD and floating-point functionality.

Configurations

AArch64 System register CPACR_EL1 is architecturally mapped to AArch32 System register CPACR.

When [HCR_EL2](#).{E2H, TGE} == {1, 1}, the fields in this register have no effect on execution at EL0 and EL1. In this case, the controls provided by [CPTR_EL2](#) are used.

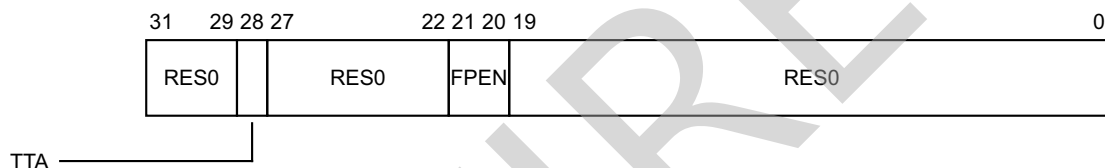
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CPACR_EL1 is a 32-bit register.

Field descriptions

The CPACR_EL1 bit assignments are:



Bits [31:29]

Reserved, RES0.

TTA, bit [28]

Traps EL0 and EL1 System register accesses to all implemented trace registers to EL1, from both Execution states.

- | | |
|---|--|
| 0 | This control does not cause any instructions to be trapped. |
| 1 | For the following settings of SCR_EL3 .NS and HCR_EL2 .TGE, EL0 and EL1 System register accesses to all implemented trace registers are trapped to EL1: <ul style="list-style-type: none"> • SCR_EL3.NS is 0, and any value of HCR_EL2.TGE. • SCR_EL3.NS is 1 and HCR_EL2.TGE is 0. When SCR_EL3 .NS is 1 and HCR_EL2 .TGE is 1: <ul style="list-style-type: none"> • If HCR_EL2.E2H is 0, Non-secure EL0 and EL1 System register accesses to all implemented trace registers are trapped to EL2. • If HCR_EL2.E2H is 1, this control does not cause any instructions to be trapped. |

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of [CPACR_EL1](#).TTA is 1.
- The ARMv8-A architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not implemented, this bit is RES0.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps EL0 and EL1 accesses to the Advanced SIMD and floating-point registers to EL1, from both Execution states.

The behavior is as follows when `SCR_EL3.NS` is 0, or when `SCR_EL3.NS` is 1 and `HCR_EL2.TGE` is 0:

- | | |
|----|--|
| 00 | This control causes any instructions in EL0 or EL1 that use the registers associated with Advanced SIMD and floating-point execution to be trapped to EL1. |
| 01 | This control causes any instructions in EL0 that use the registers associated with Advanced SIMD and floating-point execution to be trapped, but does not cause any instruction in EL1 to be trapped to EL1. |
| 10 | This control causes any instructions in EL0 or EL1 that use the registers associated with Advanced SIMD and floating-point execution to be trapped to EL1. |
| 11 | This control does not cause any instructions to be trapped. |

When `SCR_EL3.NS` is 1 and `HCR_EL2.TGE` is 1:

- If `HCR_EL2.E2H` is 0:
 - When the value of this field is 0b00, 0b01, or 0b10, this control causes any instructions in Non-secure EL0 that use the registers associated with Advanced SIMD and floating-point execution to be trapped to EL2.
 - When the value of this field is 0b11, this control does not cause any instructions to be trapped.
- If `HCR_EL2.E2H` is 1, this control does not cause any instructions to be trapped.

Writes to MVFR0, MVFR1 and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of `CPACR_EL1.FPEN` is not 11.

Bits [19:0]

Reserved, RES0.

Accessing the CPACR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CPACR_EL1	11	000	0001	0000	010
CPACR_EL12	11	101	0001	0000	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CPACR_EL1	x	x	0	-	RW	n/a	RW
CPACR_EL1	0	0	1	-	RW	RW	RW
CPACR_EL1	0	1	1	-	n/a	RW	RW
CPACR_EL1	1	0	1	-	RW	CPTR_EL2	RW
CPACR_EL1	1	1	1	-	n/a	CPTR_EL2	RW
CPACR_EL12	x	x	0	-	-	n/a	-
CPACR_EL12	0	0	1	-	-	-	-
CPACR_EL12	0	1	1	-	n/a	-	-
CPACR_EL12	1	0	1	-	-	RW	RW
CPACR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CPACR_EL1 or CPACR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [CPTR_EL2.TCPAC](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [CPTR_EL2.TCPAC](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If [CPTR_EL3.TCPAC](#)==1, accesses to this register from EL1 and EL2 are trapped to EL3.

B12.2.12 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls:

- Trapping to EL2 of access to CPACR, [CPACR_EL1](#), trace functionality, and to Advanced SIMD and floating-point functionality.
- EL2 access to trace functionality, and to Advanced SIMD and floating-point functionality.

Configurations

AArch64 System register CPTR_EL2 is architecturally mapped to AArch32 System register HCPTR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

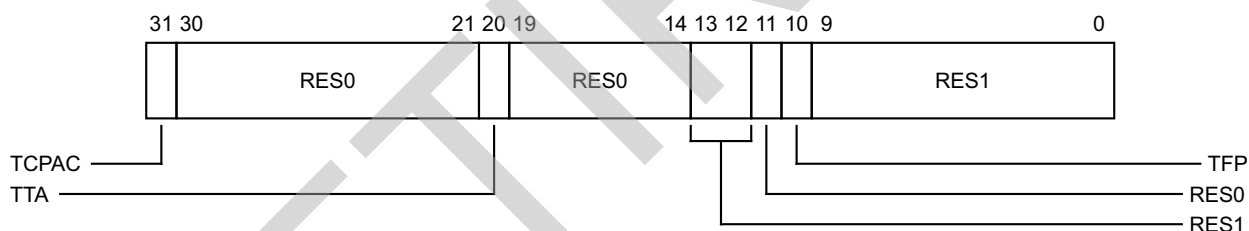
Attributes

CPTR_EL2 is a 32-bit register.

Field descriptions

The CPTR_EL2 bit assignments are:

When HCR_EL2.E2H == 0:



This format applies in all ARMv8.0 implementations.

TCPAC, bit [31]

Traps Non-secure EL1 accesses to [CPACR_EL1](#) or CPACR to EL2, from both Execution states.

- 0 This control does not cause any instructions to be trapped.
- 1 Non-secure EL1 accesses to [CPACR_EL1](#) and CPACR are trapped to EL2.

Note

[CPACR_EL1](#) and CPACR are not accessible at EL0.

Bits [30:21]

Reserved, RES0.

TTA, bit [20]

Traps Non-secure System register accesses to all implemented trace registers to EL2, from both Execution states.

- 0 This control does not cause any instructions to be trapped.
- 1 Any attempt at EL2, or Non-secure EL0 or EL1, to execute a System register access to an implemented trace register is trapped to EL2, except in the following circumstances:
 - The instruction is trapped to EL1 by [CPACR_EL1.TTA](#).

- The instruction is trapped to Undefined mode by CPACR.NSTRCDIS.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

Bits [19:14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

Bit [11]

Reserved, RES0.

TFP, bit [10]

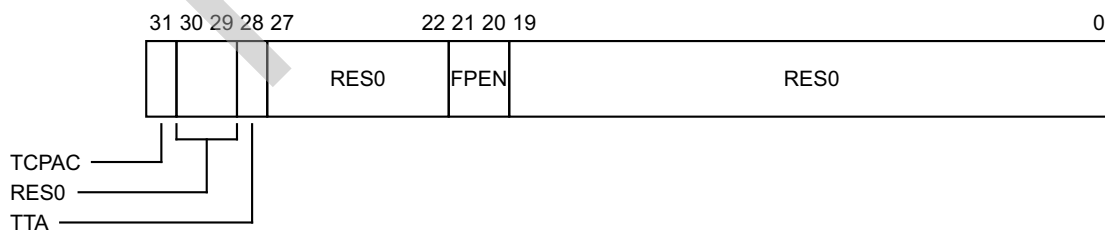
Traps Non-secure accesses to Advanced SIMD and floating-point functionality to EL2, from both Execution states.

- 0 Does not cause any instruction to be trapped.
- 1 Any attempt at EL2, or Non-secure EL0 or EL1, to execute an instruction that uses the registers associated with Advanced SIMD and floating-point execution is trapped to EL2, except in the following circumstances:
 - The instruction is trapped by CPACR_EL1.FPEN.
 - The instruction is UNDEFINED as a result of CPACR.cp10.

Bits [9:0]

Reserved, RES1.

When HCR_EL2.E2H == 1:



TCPAC, bit [31]

When HCR_EL2.TGE is 0, traps Non-secure EL1 accesses to CPACR_EL1 and CPACR to EL2, from both Execution states.

- 0 This control does not cause any instructions to be trapped.
- 1 Non-secure EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

———— **Note** ————

[CPACR_EL1](#) and CPACR are not accessible at EL0.

Bits [30:29]

Reserved, RES0.

TTA, bit [28]

Traps Non-secure System register accesses to all implemented trace registers to EL2, from both Execution states.

- | | |
|---|---|
| 0 | This control does not cause any instructions to be trapped. |
| 1 | <p>When HCR_EL2.TGE is 0, any attempt at EL2, or Non-secure EL0 or EL1, to execute a System register access to an implemented trace register is trapped to EL2, except in the following circumstances:</p> <ul style="list-style-type: none"> • The instruction is trapped by CPACR_EL1.TTA. • The instruction is trapped to Undefined mode by CPACR.NSTRCDIS. <p>When HCR_EL2.TGE is 1, any attempt at EL2, or Non-secure EL0, to execute a System register access to an implemented trace register is trapped to EL2.</p> |

———— **Note** ————

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of [CPTR_EL2.TTA](#) is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps EL2, EL0 and, when [HCR_EL2.TGE](#) is 0, EL1 accesses to the Advanced SIMD and floating-point registers to EL2, from both Execution states.

- | | |
|----|---|
| 00 | Causes any instructions in EL0, EL1, or EL2 that use the registers associated with Advanced SIMD and floating-point execution to be trapped. |
| 01 | <p>When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.</p> <p>When HCR_EL2.TGE is 1, causes instructions in EL0 that use the registers associated with Advanced SIMD and floating-point execution to be trapped.</p> |
| 10 | Causes any instructions in EL0, EL1, or EL2 that use the registers associated with Advanced SIMD and floating-point execution to be trapped. |
| 11 | This control does not cause any instructions to be trapped. |

When [HCR_EL2.TGE](#) is 0, the settings in the following controls can cause a trap that is of a higher priority:

- [CPACR_EL1.FPEN](#).
- CPACR.cp10.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not 11.

Bits [19:0]

Reserved, RES0.

Accessing the CPTR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CPTR_EL2	11	100	0001	0001	010
CPACR_EL1	11	000	0001	0000	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CPTR_EL2	x	x	0	-	-	n/a	RW
CPTR_EL2	0	0	1	-	-	RW	RW
CPTR_EL2	0	1	1	-	n/a	RW	RW
CPTR_EL2	1	0	1	-	-	RW	RW
CPTR_EL2	1	1	1	-	n/a	RW	RW
CPACR_EL1	x	x	0	-	CPACR_EL1	n/a	CPACR_EL1
CPACR_EL1	0	0	1	-	CPACR_EL1	CPACR_EL1	CPACR_EL1
CPACR_EL1	0	1	1	-	n/a	CPACR_EL1	CPACR_EL1
CPACR_EL1	1	0	1	-	CPACR_EL1	RW	CPACR_EL1
CPACR_EL1	1	1	1	-	n/a	RW	CPACR_EL1

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547. Subject to the prioritization rules:

When EL3 is implemented and is using AArch64:

- If CPTR_EL3.TCPAC==1, accesses to this register from EL2 are trapped to EL3.

RETIRED

B12.2.13 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

Purpose

When taking an exception to EL1, holds the address to return to.

Configurations

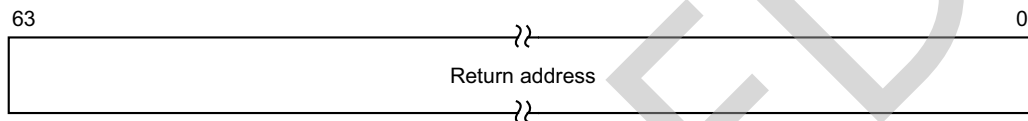
There are no configuration notes.

Attributes

ELR_EL1 is a 64-bit register.

Field descriptions

The ELR_EL1 bit assignments are:



Bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

Accessing the ELR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ELR_EL1	11	000	0100	0000	001
ELR_EL12	11	101	0100	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ELR_EL1	x	x	0	-	RW	n/a	RW
ELR_EL1	0	0	1	-	RW	RW	RW
ELR_EL1	0	1	1	-	n/a	RW	RW

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ELR_EL1	1	0	1	-	RW	ELR_EL2	RW
ELR_EL1	1	1	1	-	n/a	ELR_EL2	RW
ELR_EL12	x	x	0	-	-	n/a	-
ELR_EL12	0	0	1	-	-	-	-
ELR_EL12	0	1	1	-	n/a	-	-
ELR_EL12	1	0	1	-	-	RW	RW
ELR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic ELR_EL1 or ELR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.14 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

Purpose

When taking an exception to EL2, holds the address to return to.

Configurations

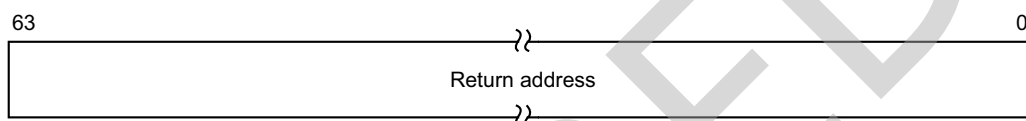
AArch64 System register ELR_EL2 is architecturally mapped to AArch32 System register ELR_hyp.

Attributes

ELR_EL2 is a 64-bit register.

Field descriptions

The ELR_EL2 bit assignments are:



Bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

Accessing the ELR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ELR_EL2	11	100	0100	0000	001
ELR_EL1	11	000	0100	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ELR_EL2	x	x	0	-	-	n/a	RW
ELR_EL2	0	0	1	-	-	RW	RW
ELR_EL2	0	1	1	-	n/a	RW	RW
ELR_EL2	1	0	1	-	-	RW	RW
ELR_EL2	1	1	1	-	n/a	RW	RW
ELR_EL1	x	x	0	-	ELR_EL1	n/a	ELR_EL1
ELR_EL1	0	0	1	-	ELR_EL1	ELR_EL1	ELR_EL1
ELR_EL1	0	1	1	-	n/a	ELR_EL1	ELR_EL1
ELR_EL1	1	0	1	-	ELR_EL1	RW	ELR_EL1
ELR_EL1	1	1	1	-	n/a	RW	ELR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic [ELR_EL2](#) or [ELR_EL1](#) are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.15 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL1.

Configurations

AArch64 System register ESR_EL1 is architecturally mapped to AArch32 System register DFSR.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

ESR_EL1 is a 32-bit register.

Field descriptions

See ESR_ELx.

Accessing the ESR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ESR_EL1	11	000	0101	0010	000
ESR_EL12	11	101	0101	0010	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ESR_EL1	x	x	0	-	RW	n/a	RW
ESR_EL1	0	0	1	-	RW	RW	RW
ESR_EL1	0	1	1	-	n/a	RW	RW
ESR_EL1	1	0	1	-	RW	ESR_EL2	RW
ESR_EL1	1	1	1	-	n/a	ESR_EL2	RW
ESR_EL12	x	x	0	-	-	n/a	-
ESR_EL12	0	0	1	-	-	-	-

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ESR_EL12	0	1	1	-	n/a	-	-
ESR_EL12	1	0	1	-	-	RW	RW
ESR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic ESR_EL1 or ESR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.16 ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL2.

Configurations

AArch64 System register ESR_EL2 is architecturally mapped to AArch32 System register HSR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

ESR_EL2 is a 32-bit register.

Field descriptions

See ESR_ELx.

Accessing the ESR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ESR_EL2	11	100	0101	0010	000
ESR_EL1	11	000	0101	0010	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ESR_EL2	x	x	0	-	-	n/a	RW
ESR_EL2	0	0	1	-	-	RW	RW
ESR_EL2	0	1	1	-	n/a	RW	RW
ESR_EL2	1	0	1	-	-	RW	RW
ESR_EL2	1	1	1	-	n/a	RW	RW
ESR_EL1	x	x	0	-	ESR_EL1	n/a	ESR_EL1
ESR_EL1	0	0	1	-	ESR_EL1	ESR_EL1	ESR_EL1

<systemr eg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ESR_EL1	0	1	1	-	n/a	ESR_EL1	ESR_EL1
ESR_EL1	1	0	1	-	ESR_EL1	RW	ESR_EL1
ESR_EL1	1	1	1	-	n/a	RW	ESR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic ESR_EL2 or ESR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

RETIRED

B12.2.17 ESR_EL3, Exception Syndrome Register (EL3)

The ESR_EL3 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL3.

Configurations

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

ESR_EL3 is a 32-bit register.

Field descriptions

See ESR_ELx.

Accessing the ESR_EL3

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ESR_EL3	11	110	0101	0010	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ESR_EL3	x	x	0	-	-	n/a	RW
ESR_EL3	0	0	1	-	-	-	RW
ESR_EL3	0	1	1	-	n/a	-	RW
ESR_EL3	1	0	1	-	-	-	RW
ESR_EL3	1	1	1	-	n/a	-	RW

B12.2.18 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous instruction or data aborts, or exceptions from a misaligned PC or a Watchpoint exception, taken to EL1.

Configurations

AArch64 System register FAR_EL1[31:0] is architecturally mapped to AArch32 System register DFAR (NS).

AArch64 System register FAR_EL1[63:32] is architecturally mapped to AArch32 System register IFAR (NS).

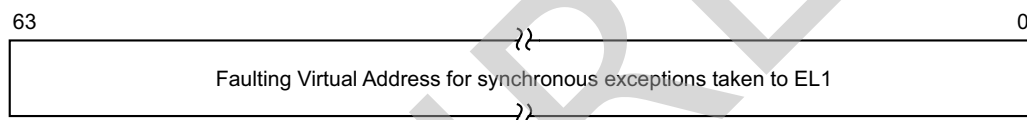
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

FAR_EL1 is a 64-bit register.

Field descriptions

The FAR_EL1 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are instruction aborts (EC 0x20 or 0x21), data aborts (EC 0x24 or 0x25), a misaligned PC exception (EC 0x22), or a Watchpoint exception (EC 0x34 or 0x35). [ESR_EL1](#).EC holds the EC value for the exception.

For a synchronous external abort other than a synchronous external abort on a translation table walk, this field is valid only if [ESR_EL1](#).FnV is 0, and the FAR_EL1 is UNKNOWN if [ESR_EL1](#).FnV is 1.

For all other exceptions taken to EL1, the FAR_EL1 is UNKNOWN.

If a memory fault that sets FAR_EL1 is generated from a data cache maintenance or DC ZVA instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL1 is taken from an Exception level that is using AArch32, the top 32 bits are all zero, unless the faulting address is generated by a load or store instruction that sequentially increments from address 0xFFFFFFFF. This is an UNPREDICTABLE condition, and in this case the upper 32-bits are set to 0x00000001.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see [Address tagging in AArch64 state on page B12-551](#).

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

Accessing the FAR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
FAR_EL1	11	000	0110	0000	000
FAR_EL12	11	101	0110	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
FAR_EL1	x	x	0	-	RW	n/a	RW
FAR_EL1	0	0	1	-	RW	RW	RW
FAR_EL1	0	1	1	-	n/a	RW	RW
FAR_EL1	1	0	1	-	RW	FAR_EL2	RW
FAR_EL1	1	1	1	-	n/a	FAR_EL2	RW
FAR_EL12	x	x	0	-	-	n/a	-
FAR_EL12	0	0	1	-	-	-	-
FAR_EL12	0	1	1	-	n/a	-	-
FAR_EL12	1	0	1	-	-	RW	RW
FAR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic FAR_EL1 or FAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2](#).TRVM==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2](#).TVM==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

RETIRED

B12.2.19 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous instruction or data aborts, or exceptions from a misaligned PC or a Watchpoint exception, taken to EL2.

Configurations

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register HDFAR.

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register HIFAR.

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register DFAR (S) when EL2 is implemented.

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register IFAR (S) when EL2 is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

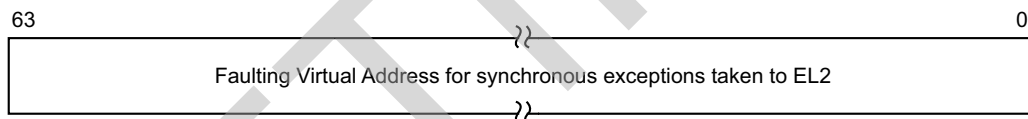
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

FAR_EL2 is a 64-bit register.

Field descriptions

The FAR_EL2 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are instruction aborts (EC 0x20 or 0x21), data aborts (EC 0x24 or 0x25), a misaligned PC exception (EC 0x22), or a Watchpoint exception (EC 0x34 or 0x35). [ESR_EL2](#).EC holds the EC value for the exception.

For a synchronous external abort other than a synchronous external abort on a translation table walk, this field is valid only if [ESR_EL2](#).FnV is 0, and the FAR_EL2 is UNKNOWN if [ESR_EL2](#).FnV is 1.

For all other exceptions taken to EL2, the FAR_EL2 is UNKNOWN.

If a memory fault that sets FAR_EL2 is generated from a data cache maintenance or DC ZVA instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL2 is taken from an Exception level that is using AArch32, the top 32-bits are all zero, unless the faulting address is generated by a load or store instruction that sequentially increments from address 0xFFFFFFFF. This is an UNPREDICTABLE condition, and in this case the upper 32-bits are set to 0x00000001.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see [Address tagging in AArch64 state on page B12-551](#).

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

Accessing the FAR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
FAR_EL2	11	100	0110	0000	000
FAR_EL1	11	000	0110	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
FAR_EL2	x	x	0	-	-	n/a	RW
FAR_EL2	0	0	1	-	-	RW	RW
FAR_EL2	0	1	1	-	n/a	RW	RW
FAR_EL2	1	0	1	-	-	RW	RW
FAR_EL2	1	1	1	-	n/a	RW	RW
FAR_EL1	x	x	0	-	FAR_EL1	n/a	FAR_EL1
FAR_EL1	0	0	1	-	FAR_EL1	FAR_EL1	FAR_EL1
FAR_EL1	0	1	1	-	n/a	FAR_EL1	FAR_EL1
FAR_EL1	1	0	1	-	FAR_EL1	RW	FAR_EL1
FAR_EL1	1	1	1	-	n/a	RW	FAR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic FAR_EL2 or FAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.20 HACR_EL2, Hypervisor Auxiliary Control Register

The HACR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

Note

ARM recommends the values in this register do not cause unnecessary traps to EL2 when [HCR_EL2](#).{E2H, TGE} = {1, 1}.

Configurations

AArch64 System register HACR_EL2 is architecturally mapped to AArch32 System register HACR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

HACR_EL2 is a 32-bit register.

Field descriptions

The HACR_EL2 bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing the HACR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
HACR_EL2	11	100	0001	0001	111

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
HACR_EL2	x	x	0	-	-	n/a	RW
HACR_EL2	0	0	1	-	-	RW	RW
HACR_EL2	0	1	1	-	n/a	RW	RW
HACR_EL2	1	0	1	-	-	RW	RW
HACR_EL2	1	1	1	-	n/a	RW	RW

RETIRED

B12.2.21 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to EL2.

Configurations

AArch64 System register HCR_EL2[31:0] is architecturally mapped to AArch32 System register HCR.

AArch64 System register HCR_EL2[63:32] is architecturally mapped to AArch32 System register HCR2.

If EL2 is not implemented, this register is RES0 from EL3.

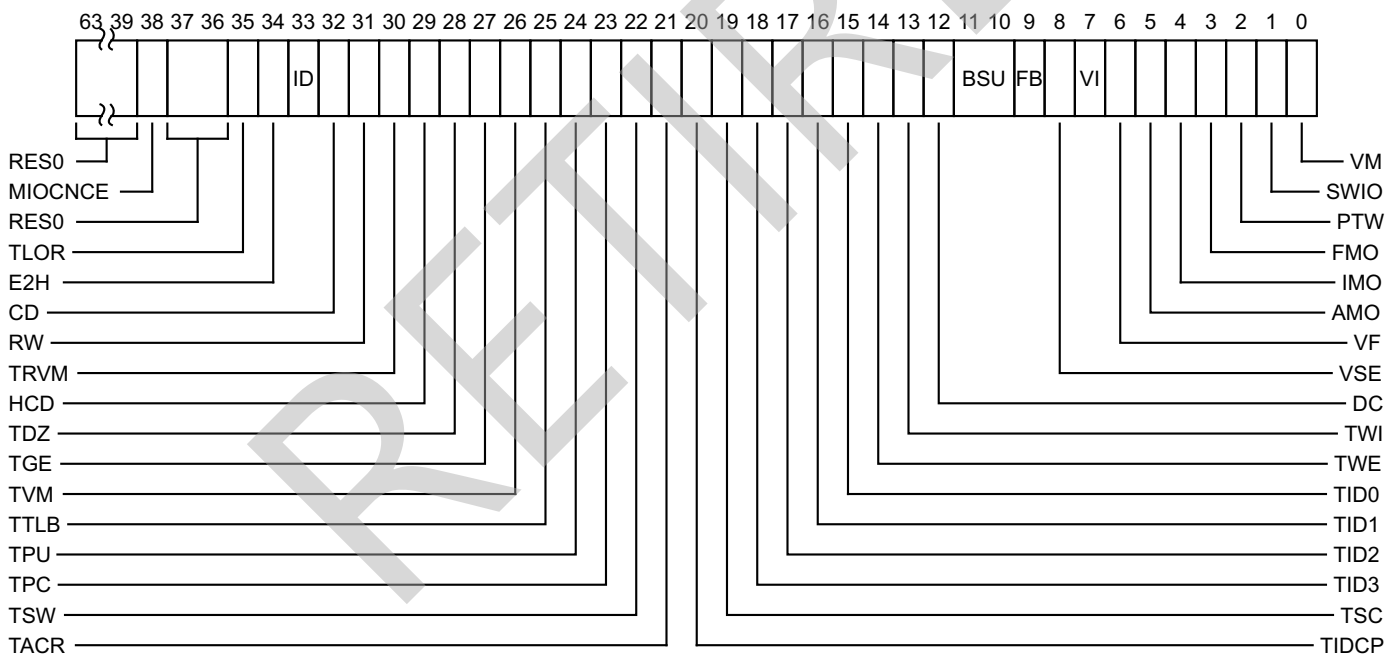
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

HCR_EL2 is a 64-bit register.

Field descriptions

The HCR_EL2 bit assignments are:



Bits [63:39]

Reserved, RES0.

MIOCNCEN, bit [38]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the Non-secure EL1&0 translation regime.

0 For the Non-secure EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

- 1 For the Non-secure EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information see [Mismatched memory attributes on page B12-545](#).

This field can be implemented as RAZ/WI.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

Bits [37:36]

Reserved, RES0.

TLOR, bit [35] (In ARMv8.1)

Trap LOR registers. Traps accesses to the [LORSA_EL1](#), [LOREA_EL1](#), [LORN_EL1](#), [LORC_EL1](#), and [LORID_EL1](#) registers from Non-secure EL1 to EL2.

0 This control has no effect on Non-secure EL1 accesses to the LOR registers.

1 Non-secure EL1 accesses to the LOR registers are trapped to EL2.

When [HCR_EL2](#).TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Bit [35] (In ARMv8.0)

Reserved, RES0.

E2H, bit [34] (In ARMv8.1)

EL2 Host. Enables a configuration where a Host Operating System is running in EL2, and the Host Operating System's applications are running in EL0.

0 EL2 is running a hypervisor.

1 EL2 is running a Host Operating System.

For information on the behavior of this bit see [Behavior of HCR_EL2.E2H on page B8-60](#).

This bit is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

Bit [34] (In ARMv8.0)

Reserved, RES0.

ID, bit [33]

Stage 2 Instruction access cacheability disable. For the Non-secure EL1&0 translation regime, when [HCR_EL2.VM](#)==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

0 This control has no effect on stage 2 of the Non-secure EL1&0 translation regime.

1 For the Non-secure EL1&0 translation regime, forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

This bit has no effect on the EL2 or EL3 translation regimes.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

CD, bit [32]

Stage 2 Data access cacheability disable. For the Non-secure EL1&0 translation regime, when `HCR_EL2.VM==1`, this control forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

- 0 This control has no effect on stage 2 of the Non-secure EL1&0 translation regime for data accesses and translation table walks.
- 1 For the Non-secure EL1&0 translation regime, forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

In an implementation that includes EL3, when the value of `SCR_EL3.NS` is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of `HCR_EL2`.

This bit has no effect on the EL2 or EL3 translation regimes.

In ARMv8.1, if `HCR_EL2.{E2H, TGE}` is set to `{1, 1}`, the PE ignores the value of this field for all purposes other than a direct read of this field.

RW, bit [31]

Execution state control for lower Exception levels:

- 0 Lower levels are all AArch32.
- 1 The Execution state for EL1 is AArch64. The Execution state for EL0 is determined by the current value of `PSTATE.nRW` when executing at EL0.

If all lower Exception levels cannot use AArch32 then this bit is RAO/WI.

In an implementation that includes EL3, when `SCR_EL3.NS==0`, the PE behaves as if this bit has the same value as the `SCR_EL3.RW` bit for all purposes other than a direct read or write access of `HCR_EL2`.

The RW bit is permitted to be cached in a TLB.

In ARMv8.1, if `HCR_EL2.{E2H, TGE}` is set to `{1, 1}`, this field behaves as 1 for all purposes other than a direct read of the value of this bit.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps Non-secure EL1 reads of the virtual memory control registers to EL2, from both Execution states. The registers for which read accesses are trapped are as follows:

Non-secure EL1 using AArch64: `SCTLR_EL1`, `TTBR0_EL1`, `TTBR1_EL1`, `TCR_EL1`, `ESR_EL1`, `FAR_EL1`, `AFSR0_EL1`, `AFSR1_EL1`, `MAIR_EL1`, `AMAIR_EL1`, `CONTEXTIDR_EL1`.

Non-secure EL1 using AArch32: `SCTLR`, `TTBR0`, `TTBR1`, `TTBCR`, `DACR`, `DFSR`, `IFSR`, `DFAR`, `IFAR`, `ADFSR`, `AIFSR`, `PRRR`, `NMRR`, `MAIR0`, `MAIR1`, `AMAIR0`, `AMAIR1`, `CONTEXTIDR`.

- 0 This control has no effect on Non-secure EL1 read accesses to Virtual Memory controls.
- 1 Non-secure EL1 read accesses to the specified Virtual Memory controls are trapped to EL2.

In an implementation that includes EL3, when the value of `SCR_EL3.NS` is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of `HCR_EL2`.

When `HCR_EL2.TGE` is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

HCD, bit [29]

HVC instruction disable. Disables Non-secure state execution of HVC instructions, from both Execution states.

- 0 HVC instruction execution is enabled at EL2 and Non-secure EL1.
- 1 HVC instructions are UNDEFINED at EL2 and Non-secure EL1. Any resulting exception is taken to the Exception level at which the HVC instruction is executed.

Note

HVC instructions are always UNDEFINED at EL0.

This bit is only implemented if EL3 is not implemented. Otherwise, it is RES0.

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TDZ, bit [28]

Trap DC ZVA instructions. Traps Non-secure EL0 and EL1 execution of DC ZVA instructions to EL2, from AArch64 state only.

- | | |
|---|--|
| 0 | This control has no effect on the Non-secure EL0 and EL1 execution of DC ZVA instructions. |
| 1 | Any attempt to execute a DC ZVA instruction at Non-secure EL0 using AArch64 or Non-secure EL1 using AArch64 is trapped to EL2.
Reading the DCZID_EL0 returns a value that indicates that DC ZVA instructions are not supported. |

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2.{E2H, TGE}](#) is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

TGE, bit [27]

Trap General Exceptions, from Non-secure EL0.

- | | |
|---|--|
| 0 | This control has no effect on execution at EL0. |
| 1 | When the value of SCR_EL3.NS is 0, this control has no effect on execution at EL0.
When the value of SCR_EL3.NS is 1, in all cases: <ul style="list-style-type: none"> • All exceptions that would be routed to EL1 are routed to EL2. • The SCTLR_EL1.M field, or the SCTLR.M field if EL1 is using AArch32, is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1 or SCTLR. • All virtual interrupts are disabled. • Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled. • An exception return to EL1 is treated as an illegal exception return. |

When the value of [SCR_EL3.NS](#) is 1 and the value of [HCR_EL2.E2H](#) is 0, additionally:

- The [HCR_EL2.{FMO, IMO, AMO}](#) fields are treated as being 1 for all purposes other than a direct read or write access of [HCR_EL2](#).
- The [MDCR_EL2.{TDRA, TDOSA, TDA}](#) fields are treated as being 1 for all purposes other than returning the result of a direct read of [MDCR_EL2](#).

For information on the behavior of this bit when E2H is 1, see *Behavior of [HCR_EL2.E2H](#)* on page B8-60.

[HCR_EL2.TGE](#) must not be cached in a TLB.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

TVM, bit [26]

Trap Virtual Memory controls. Traps Non-secure EL1 writes to the virtual memory control registers to EL2, from both Execution states. The registers for which write accesses are trapped are as follows:

Non-secure EL1 using AArch64: [SCTLR_EL1](#), [TTBR0_EL1](#), [TTBR1_EL1](#), [TCR_EL1](#), [ESR_EL1](#), [FAR_EL1](#), [AFSR0_EL1](#), [AFSR1_EL1](#), [MAIR_EL1](#), [AMAIR_EL1](#), [CONTEXTIDR_EL1](#).

Non-secure EL1 using AArch32: [SCTLR](#), TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFS, AIFS, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

- | | |
|---|---|
| 0 | This control has no effect on Non-secure EL1 write accesses to EL1 virtual memory control registers. |
| 1 | Non-secure EL1 write accesses to the specified EL1 virtual memory control registers are trapped to EL2. |

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps Non-secure EL1 execution of TLB maintenance instructions to EL2, from both Execution states. This applies to the following instructions:

Non-secure EL1 using AArch64: [TLBI VMALLE1IS](#), [TLBI VAE1IS](#), [TLBI ASIDE1IS](#), [TLBI VAAE1IS](#), [TLBI VALE1IS](#), [TLBI VAALE1IS](#), [TLBI VMALLE1](#), [TLBI VAE1](#), [TLBI ASIDE1](#), [TLBI VAAE1](#), [TLBI VALE1](#), [TLBI VAALE1](#).

Non-secure EL1 using AArch32: [TLBIALLIS](#), [TLBIMVAIS](#), [TLBIASIDIS](#), [TLBIMVAAIS](#), [TLBIMVALIS](#), [TLBIMVAALIS](#), [ITLBIALL](#), [ITLBIMVA](#), [ITLBIASID](#), [DTLBIALL](#), [DTLBIASID](#), [TLBIALL](#), [TLBIMVA](#), [TLBIASID](#), [TLBIMVAA](#), [TLBIMVAL](#), [TLBIMVAAL](#).

- | | |
|---|--|
| 0 | This control has no effect on Non-secure EL1 execution of TLB maintenance instructions. |
| 1 | Non-secure EL1 execution of the specified TLB maintenance instructions are trapped to EL2. |

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at Non-secure EL1 or EL0 using AArch64, and at Non-secure EL1 using AArch32, to EL2. This applies to the following instructions:

Non-secure EL0 using AArch64: [IC IVAU](#), [DC CVAU](#). However, if the value of [SCTLR_EL1.UCI](#) is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.

Non-secure EL1 using AArch64: [IC IVAU](#), [IC IALLU](#), [IC IALLUIS](#), [DC CVAU](#).

Non-secure EL1 using AArch32: [ICIMVAU](#), [IC IALLU](#), [IC IALLUIS](#), [DCCMVAU](#).

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- [IC IALLUIS](#) and [IC IALLU](#) are always UNDEFINED at EL0 using AArch64.
- [ICIMVAU](#), [IC IALLU](#), [IC IALLUIS](#), and [DCCMVAU](#) are always UNDEFINED at EL0 using AArch32.

- | | |
|---|--|
| 0 | This control has no effect on the execution of cache maintenance instructions. |
| 1 | Non-secure execution of the specified instructions is trapped to EL2. |

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

TPC, bit [23]

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps execution of those cache maintenance instructions at Non-secure EL1 or EL0 using AArch64, and at Non-secure EL1 using AArch32, to EL2. This applies to the following instructions:

Non-secure EL0 using AArch64: DC CIVAC, DC CVAC. However, if the value of [SCTLR_EL1](#).UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.

Non-secure EL1 using AArch64: DC IVAC, DC CIVAC, DC CVAC.

Non-secure EL1 using AArch32: DCIMVAC, DCCIMVAC, DCCMVAC.

————— Note —————

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- DC IVAC is always UNDEFINED at EL0 using AArch64.
- DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.

0 This control has no effect on the execution of cache maintenance instructions.

1 Non-secure execution of the specified instructions is trapped to EL2.

In an implementation that includes EL3, when the value of [SCR_EL3](#).NS is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at Non-secure EL1 using AArch64, and at Non-secure EL1 using AArch32, to EL2. This applies to the following instructions:

Non-secure EL1 using AArch64: DC ISW, DC CSW, DC CISW.

Non-secure EL1 using AArch32: DCISW, DCCSW, DCCISW.

————— Note —————

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0 This control has no effect on the execution of cache maintenance instructions.

1 Non-secure execution of the specified instructions is trapped to EL2.

In an implementation that includes EL3, when the value of [SCR_EL3](#).NS is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2](#).TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TACR, bit [21]

Trap Auxiliary Control Registers. Traps Non-secure EL1 accesses to the Auxiliary Control Registers to EL2, from both Execution states. This applies to the following register accesses:

- Non-secure EL1 using AArch64: [ACTLR_EL1](#).
- Non-secure EL1 using AArch32: ACTLR and, if implemented, ACTLR2.

0 This control has no effect on Non-secure EL1 accesses to the Auxiliary Control Registers.

1 Non-secure EL1 accesses to the specified registers are trapped to EL2.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps Non-secure EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2. This applies to the following register accesses:

AArch64: The following reserved encoding spaces:

- IMPLEMENTATION DEFINED system operations, which are accessed using SYS and SYSL, with CRm == {13, 15}.
- IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.

AArch32: MCR and MRC instructions accessing the following encodings:

- All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
- All coproc==p15, CRn==c10, opc1 == {0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
- All coproc==p15, CRn==c11, opc1 == {0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

When the value of [HCR_EL2.TIDCP](#) is 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure EL0 is trapped to EL2. If it is not, then it is UNDEFINED, and any attempt to access it from Non-secure EL0 generates an exception that is taken to EL1.

- | | |
|---|--|
| 0 | This control has no effect on the encodings reserved for IMPLEMENTATION DEFINED functionality. |
| 1 | Non-secure EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2. |

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TSC, bit [19]

Trap SMC instructions. Traps Non-secure EL1 execution of SMC instructions to EL2, from both Execution states.

- | | |
|---|---|
| 0 | This control has no effect on the execution of SMC instructions. |
| 1 | Any attempt to execute an SMC instruction at Non-secure EL1 using AArch64 or Non-secure EL1 using AArch32 is trapped to EL2, regardless of the value of SCR_EL3.SMD . |

In AArch32 state, the ARMv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

If EL3 is not implemented, this bit is RES0.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TID3, bit [18]

Trap ID group 3. Traps Non-secure EL1 reads of the following registers to EL2:

AArch64: ID_PFR0_EL1, ID_PFR1_EL1, [ID_DFR0_EL1](#), ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, [ID_MMFR3_EL1](#), ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, [ID_ISAR5_EL1](#), MVFR0_EL1, MVFR1_EL1, MVFR2_EL1, ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, [ID_AA64DFR0_EL1](#),

ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64AFR0_EL1, ID_AA64AFR1_EL1, and ID_MMFR4_EL1, except that if ID_MMFR4_EL1 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4_EL1 are trapped.

It is IMPLEMENTATION DEFINED whether this field traps MRS accesses to encodings in the following range that are not already mentioned in this field description:

- Op0 == 3, op1 == 0, CRn == c0, CRm == {c2-c7}, op2 == {0-7}.

AArch32: ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, MVFR0, MVFR1, MVFR2, and ID_MMFR4, except that if ID_MMFR4 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 are trapped.

MRC access to any of the following encodings are also trapped:

- coproc==p15, opc1 == 0, CRn == c0, CRm == {c3-c7}, opc2 == {0,1}.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c3, opc2 == 2.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c5, opc2 == {4,5}.

It is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to the following encodings:

- coproc==p15, opc1 == 0, CRn == c0, CRm == c2, opc2 == 7.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c3, opc2 == {3-7}.
- coproc==p15, opc1 == 0, CRn == c0, CRm == {c4, c6, c7}, opc2 == {2-7}.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c5, opc2 == {2, 3, 6, 7}.

0 This control has no effect on Non-secure EL1 reads of the ID group 3 registers.

1 The specified Non-secure EL1 read accesses to ID group 3 registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of HCR_EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TID2, bit [17]

Trap ID group 2. Traps the following register accesses to EL2:

AArch64:

- Non-secure EL1 reads of CTR_EL0, CCSIDR_EL1, CLIDR_EL1, and CSSELR_EL1.
- Non-secure EL0 reads of CTR_EL0, except that if the value of SCTLRL_EL1.UCT is 0 then EL0 reads of CTR_EL0 are UNDEFINED and any resulting exception takes precedence over this trap.
- Non-secure EL1 writes to CSSELR_EL1.

AArch32:

- Non-secure EL1 reads of the CTR, CCSIDR, CLIDR, and CSSELR.
- Non-secure EL1 writes to the CSSELR.

0 This control has no effect on Non-secure EL1 and EL0 accesses to the ID group 2 registers.

1 The specified Non-secure EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of HCR_EL2.

In ARMv8.1, if HCR_EL2.{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

TID1, bit [16]

Trap ID group 1. Traps Non-secure EL1 reads of the following registers are trapped to EL2:

AArch64: REVIDR_EL1, AIDR_EL1.

AArch32: TCMTR, TLBTR, REVIDR, AIDR.

0 This control has no effect on Non-secure EL1 reads of the ID group 1 registers.

1 The specified Non-secure EL1 read accesses to ID group 1 registers are trapped to EL2.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

TID0, bit [15]

Trap ID group 0. Traps the following register accesses to EL2:

AArch64: None.

AArch32:

- Non-secure EL1 reads of the JIDR.
- If the JIDR is RAZ from Non-secure EL0, Non-secure EL0 of the JIDR.
- Non-secure EL1 reads of the FPSID.

Note

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then any resulting exception takes precedence over this trap.
- The FPSID is not accessible at EL0 using AArch32.
- Writes to the FPSID are ignored, and not trapped by this control.

0 This control has no effect on Non-secure EL1 reads of the ID group 0 registers.

1 The specified Non-secure EL1 read accesses to ID group 0 registers are trapped to EL2.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2.{E2H, TGE}](#) is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

TWE, bit [14]

Traps Non-secure EL0 and EL1 execution of WFE instructions to EL2, from both Execution states.

0 This control has no effect on the execution of WFE instructions at Non-secure EL0 or Non-secure EL1.

1 Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of [SCTLR_EL1.nTWE](#) is 0, the trap of EL0 execution to EL1 takes precedence over this trap.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In ARMv8.1, if [HCR_EL2.{E2H, TGE}](#) is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

TWI, bit [13]

Traps Non-secure EL0 and EL1 execution of WFI instructions to EL2, from both Execution states.

- | | |
|---|---|
| 0 | This control has no effect on the execution of WFI instructions at Non-secure EL1 or Non-secure EL0. |
| 1 | Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR_EL1.nTWI is 0, the trap of EL0 execution to EL1 takes precedence over this trap. |

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

———— Note ————

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

DC, bit [12]

Default Cacheability.

- | | |
|---|--|
| 0 | This control has no effect on the Non-secure EL1&0 translation regime. |
| 1 | <p>In Non-secure state:</p> <ul style="list-style-type: none"> • When EL1 is using AArch64, the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1. • When EL1 is using AArch32, the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of SCTLR. • The PE behaves as if the value of the HCR_EL2.VM field is 1 for all purposes other than returning the value of a direct read of HCR_EL2. • The memory type produced by stage 1 of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read- |

This field has no effect on the EL2 and EL3 translation regimes.

This field is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from Non-secure EL1 or Non-secure EL0:

- | | |
|----|-----------------|
| 00 | No effect |
| 01 | Inner Shareable |
| 10 | Outer Shareable |
| 11 | Full system |

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this field behaves as 0b00 for all purposes other than a direct read of the value of this bit.

FB, bit [9]

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from Non-secure EL1:

AArch32: BPIALL, TLBIALl, TLBIMVA, TLBIASID, DTLBIALl, DTLBIMVA, DTLBIASID, ITLBIALl, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALlU, TLBIMVAL, TLBIMVAAL.

AArch64: [TLBI VMALLE1](#), [TLBI VAE1](#), [TLBI ASIDE1](#), [TLBI VAAE1](#), [TLBI VALE1](#), [TLBI VAALE1](#), IC IALLU.

0 This field has no effect on the operation of the specified instructions.

1 When one of the specified instruction is executed at Non-secure EL1, the instruction is broadcast within the Inner Shareable shareability domain.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2](#).TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

VSE, bit [8]

Virtual System Error or Asynchronous Abort.

0 This mechanism is not making a virtual System Error or Asynchronous Abort pending.

1 A virtual System Error or Asynchronous Abort is pending because of this mechanism.

The virtual System Error or Asynchronous Abort is only enabled when the value of [HCR_EL2.AMO](#) bit is 1.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

VI, bit [7]

Virtual IRQ Interrupt.

0 This mechanism is not making a virtual IRQ pending.

1 A virtual IRQ is pending because of this mechanism.

The virtual IRQ is only enabled when the value of [HCR_EL2.IMO](#) bit is 1.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

VF, bit [6]

Virtual FIQ Interrupt.

0 This mechanism is not making a virtual FIQ pending.

1 A virtual FIQ is pending because of this mechanism.

The virtual FIQ is only enabled when the value of [HCR_EL2.FMO](#) bit is 1.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

AMO, bit [5]

Asynchronous External Abort and SError Interrupt routing.

0 When executing at Non-secure Exception levels below EL2, physical Asynchronous External Aborts and SError Interrupts are not taken to EL2.

When executing at EL2 using AArch64, physical Asynchronous External Aborts and SError Interrupts are not taken unless they are routed to EL3 by the [SCR_EL3.EA](#) bit.

Virtual Asynchronous External Aborts and SError Interrupts interrupts are disabled.

1 When executing at any Exception level in Non-secure state:

- Physical Asynchronous External Aborts and SError Interrupts are taken to EL2 unless they are routed to EL3.
- Virtual Asynchronous External Aborts and SError Interrupts are enabled.

If the value of HCR_EL2.TGE is 1:

- In ARMv8.0, or in ARMv8.1 when [HCR_EL2.E2H](#) is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
- In ARMv8.1 when [HCR_EL2.E2H](#) is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see [Asynchronous exception routing on page B12-550](#).

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of HCR_EL2.

IMO, bit [4]

Physical IRQ Routing.

0 When executing at Non-secure Exception levels below EL2, physical IRQ interrupts are not taken to EL2.
When executing at EL2 using AArch64, physical IRQ interrupts are not taken unless they are routed to EL3 by the [SCR_EL3.IRQ](#) bit.
Virtual IRQ interrupts are disabled.

1 When executing at any Exception level in Non-secure state:

- Physical IRQ interrupts are taken to EL2 unless they are routed to EL3.
- Virtual IRQ interrupts are enabled in Non-secure state.

If the value of HCR_EL2.TGE is 1:

- In ARMv8.0, or in ARMv8.1 when [HCR_EL2.E2H](#) is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
- In ARMv8.1 when [HCR_EL2.E2H](#) is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see [Asynchronous exception routing on page B12-550](#).

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of HCR_EL2.

FMO, bit [3]

Physical FIQ Routing.

0 When executing at Non-secure Exception levels below EL2, physical FIQ interrupts are not taken to EL2.
When executing at EL2 using AArch64, physical FIQ interrupts are not taken unless they are routed to EL3 by the [SCR_EL3.FIQ](#) bit.
Virtual FIQ interrupts are disabled.

1 When executing at any Exception level in Non-secure state:

- Physical FIQ interrupts are taken to EL2 unless they are routed to EL3.
- Virtual FIQ interrupts are enabled in Non-secure state.

If the value of HCR_EL2.TGE is 1:

- In ARMv8.0, or in ARMv8.1 when [HCR_EL2.E2H](#) is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
- In ARMv8.1 when [HCR_EL2.E2H](#) is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see [Asynchronous exception routing on page B12-550](#).

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

PTW, bit [2]

Protected Table Walk. In the Non-secure EL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs then the value of this bit determines the behavior:

- 0 The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.
- 1 The memory access generates a stage 2 Permission fault.

This field is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

SWIO, bit [1]

Set/Way Invalidation Override. Causes Non-secure EL1 execution of the data cache invalidate by set/way instructions to be treated as data cache clean and invalidate by set/way:

- 0 This control has no effect on the operation of data cache invalidate by set/way instructions.
- 1 Data cache invalidate by set/way instructions operate as data cache clean and invalidate by set/way.

When the value of this bit is 1:

AArch32: DCISW is executed as DCCISW.

AArch64: DC ISW is executed as DC CISW.

This bit can be implemented as RES1.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime. Possible values of this bit are:

- 0 Non-secure EL1&0 stage 2 address translation disabled.
- 1 Non-secure EL1&0 stage 2 address translation enabled.

When the value of this bit is 1, data cache invalidate instructions executed at Non-secure EL1 operate as data cache clean and invalidate instructions. For the invalidate by set/way instruction this behavior applies regardless of the value of the [HCR_EL2.SWIO](#) bit.

This bit is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of [SCR_EL3.NS](#) is 0 the PE behaves as if this field is 0 for all purposes other than a direct read or write access of [HCR_EL2](#).

In ARMv8.1, if [HCR_EL2.{E2H, TGE}](#) is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Accessing the HCR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
HCR_EL2	11	100	0001	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
HCR_EL2	x	x	0	-	-	n/a	RW
HCR_EL2	0	0	1	-	-	RW	RW
HCR_EL2	0	1	1	-	n/a	RW	RW
HCR_EL2	1	0	1	-	-	RW	RW
HCR_EL2	1	1	1	-	n/a	RW	RW

B12.2.22 HSTR_EL2, Hypervisor System Trap Register

The HSTR_EL2 characteristics are:

Purpose

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower in AArch32, to the System register in the coproc == 1111 encoding space, by the CRn value used to access the register using MCR or MRC instruction. When the register is accessible using an MCRR or MRRC instruction, this is the CRm value used to access the register.

Configurations

AArch64 System register HSTR_EL2 is architecturally mapped to AArch32 System register HSTR.

If EL2 is not implemented, this register is RES0 from EL3.

If no Exception level can use AArch32, then this register is RES0

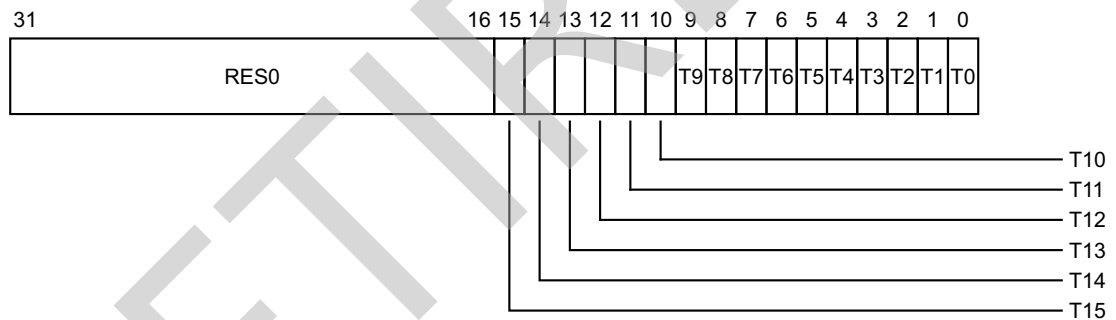
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

HSTR_EL2 is a 32-bit register.

Field descriptions

The HSTR_EL2 bit assignments are:



Bits [31:16]

Reserved, RES0.

T<n>, bit [n], for n = 0 to 15

Fields T14 and T4 are RES0.

The remaining fields control whether Non-secure EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the System registers in the coproc == 1111 encoding space are trapped to Hyp mode:

- 0 This control has no effect on Non-secure EL0 or EL1 accesses to System registers.
- 1 Any Non-secure EL1 MCR, MRC access with coproc == 1111 and CRn == <n> is trapped to Hyp mode if the access is not UNDEFINED when the value of this field is 0.
Any Non-secure EL1 MCRR, MRRC access with coproc == 1111 and CRm == <n> is trapped to Hyp mode if the access is not UNDEFINED when the value of this field is 0.

For example, when HSTR_EL2.T7 is 1:

- Any 32-bit access from a Non-secure EL1 mode, using an MCR or MRC instruction with coproc set to 1111 and <CRn> set to c7, and that is not UNDEFINED when HSTR_EL2.T7 is 0, is trapped to Hyp mode.

- Any 64-bit access from a Non-secure EL1 mode, using an MCRR or MRRC instruction with coproc set to 1111 and <CRm> set to c7, and that is not UNDEFINED when HSTR_EL2.T7 is 0, is trapped to Hyp mode.

In ARMv8.1, if HCR_EL2.{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

Accessing the HSTR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
HSTR_EL2	11	100	0001	0001	011

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
HSTR_EL2	x	x	0	-	-	n/a	RW
HSTR_EL2	0	0	1	-	-	RW	RW
HSTR_EL2	0	1	1	-	n/a	RW	RW
HSTR_EL2	1	0	1	-	-	RW	RW
HSTR_EL2	1	1	1	-	n/a	RW	RW

B12.2.23 ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0

The ID_AA64DFR0_EL1 characteristics are:

Purpose

Provides top level information about the debug system in AArch64.

Configurations

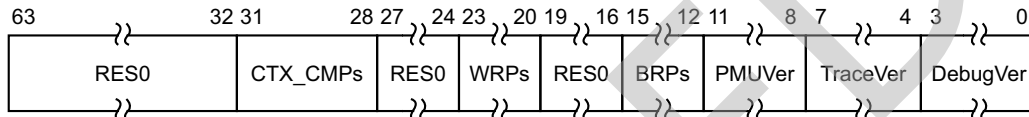
The external register EDDFR gives information from this register.

Attributes

ID_AA64DFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64DFR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

CTX_CMPs, bits [31:28]

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1. The value of 0b0000 is reserved.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1. The value of 0b0000 is reserved.

PMUVer, bits [11:8]

Performance Monitors extension version. Indicates whether System register interface to Performance Monitors extension is implemented. Defined values are:

- | | |
|------|---|
| 0000 | Performance Monitors extension System registers not implemented. |
| 0001 | Performance Monitors extension System registers implemented, PMUv3. |
| 0100 | Performance Monitors extension System registers implemented, PMUv3, with a 16-bit evtCount field, and if EL2 is implemented, the addition of the MDCR_EL2.HPMD bit. |
| 1111 | IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. |

All other values are reserved.

In ARMv8-A the permitted values are 0b0000, 0b0001 and 0b1111.

In ARMv8.1 the permitted values are 0b0000, 0b0100 and 0b1111.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace macrocell is implemented. Defined values are:

- 0000 Trace macrocell System registers not implemented.
- 0001 Trace macrocell System registers implemented.

All other values are reserved.

A value of 0b0000 only indicates that no System register interface to a trace macrocell is implemented. A trace macrocell might nevertheless be implemented without a System register interface.

DebugVer, bits [3:0]

Debug architecture version. Indicates presence of ARMv8 debug architecture.

- 0110 ARMv8 debug architecture.
- 0111 ARMv8 debug architecture with Virtualization Host Extensions.

All other values are reserved.

In ARMv8-A the only permitted value is 0b0110.

In ARMv8.1 the only permitted value is 0b0111.

Accessing the ID_AA64DFR0_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ID_AA64DFR0_EL1	11	000	0000	0101	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ID_AA64DFR0_EL1	x	x	0	-	RO	n/a	RO
ID_AA64DFR0_EL1	x	0	1	-	RO	RO	RO
ID_AA64DFR0_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TID3](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

RETIRED

B12.2.24 ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0

The ID_AA64ISAR0_EL1 characteristics are:

Purpose

Provides information about the instructions implemented in AArch64 state.

Configurations

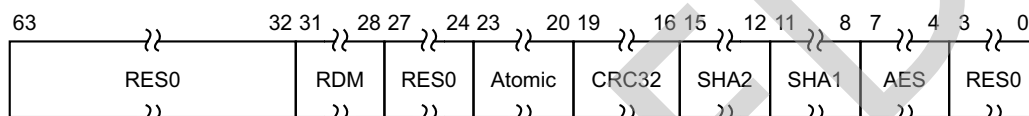
There are no configuration notes.

Attributes

ID_AA64ISAR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR0_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

RDM, bits [31:28] (In ARMv8.1)

SQRDMLAH and SQRDMLSH instructions in AArch64. Defined values are:

0000 No SQRDMLAH and SQRDMLSH instructions implemented.

0001 SQRDMLAH and SQRDMLSH instructions implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

In ARMv8.1 the only permitted value is 0001.

Bits [31:28] (In ARMv8.0)

Reserved, RES0.

Bits [27:24]

Reserved, RES0.

Atomic, bits [23:20] (In ARMv8.1)

Atomic instructions in AArch64. Defined values are:

0000 No Atomic instructions implemented.

0010 LDADD, LDCLR, LDEOR, LDSET, LDSMAX, LDSMIN, LDUMAX, LDUMIN, CAS, CASP, and SWP instructions implemented.

All other values are reserved.

In ARMv8.1 the only permitted value is 0010.

Bits [23:20] (In ARMv8.0)

Reserved, RES0.

CRC32, bits [19:16]

CRC32 instructions in AArch64. Defined values are:

0000 No CRC32 instructions implemented.

0001 CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX instructions implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In ARMv8.1 the only permitted value is 0001.

SHA2, bits [15:12]

SHA2 instructions in AArch64. Defined values are:

0000 No SHA2 instructions implemented.

0001 SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 instructions implemented.

All other values are reserved.

SHA1, bits [11:8]

SHA1 instructions in AArch64. Defined values are:

0000 No SHA1 instructions implemented.

0001 SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 instructions implemented.

All other values are reserved.

AES, bits [7:4]

AES instructions in AArch64. Defined values are:

0000 No AES instructions implemented.

0001 AESE, AESD, AESMC, and AESIMC instructions implemented.

0010 As for 0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

Bits [3:0]

Reserved, RES0.

Accessing the ID_AA64ISAR0_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ID_AA64ISAR0_EL1	11	000	0000	0110	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ID_AA64ISAR0_EL1	x	x	0	-	RO	n/a	RO
ID_AA64ISAR0_EL1	x	0	1	-	RO	RO	RO
ID_AA64ISAR0_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

B12.2.25 ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1

The ID_AA64MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64.

Configurations

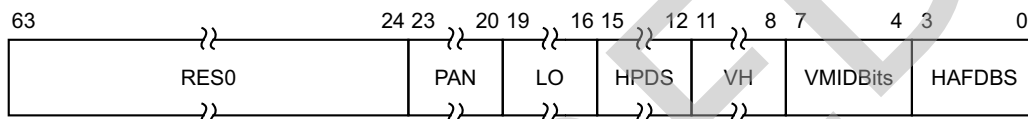
There are no configuration notes.

Attributes

ID_AA64MMFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR1_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

PAN, bits [23:20] (In ARMv8.1)

Privileged Access Never. Indicates support for the PAN bit in PSTATE, [SPSR_EL1](#), [SPSR_EL2](#), [SPSR_EL3](#), and [DSPSR_EL0](#). Defined values are:

0000 PAN not supported.

0001 PAN supported.

All other values are reserved.

In ARMv8.1 the only permitted value is 0001.

Bits [23:20] (In ARMv8.0)

Reserved, RES0.

LO, bits [19:16] (In ARMv8.1)

LORegions. Indicates support for LORegions. Defined values are:

0000 LORegions not supported.

0001 LORegions supported.

All other values are reserved.

In ARMv8.1 the only permitted value is 0001.

Bits [19:16] (In ARMv8.0)

Reserved, RES0.

HPDS, bits [15:12] (In ARMv8.1)

Hierarchical permission disable bits in translation tables. Defined values are:

0000 Not supported.

0001 Disabling of hierarchical controls supported with the [TCR_EL1](#).HPD0, [TCR_EL1](#).HPD1, [TCR_EL2](#).HPD, and [TCR_EL3](#).HPD bits.

All other values are reserved.

In ARMv8.1 the only permitted value is 0001.

Bits [15:12] (In ARMv8.0)

Reserved, RES0.

VH, bits [11:8] (In ARMv8.1)

Virtualization Host Extensions. Defined values are:

0000 Virtualization Host Extensions not supported.

0001 Virtualization Host Extensions supported.

All other values are reserved.

In ARMv8.1 the only permitted value is 0001.

Bits [11:8] (In ARMv8.0)

Reserved, RES0.

VMIDBits, bits [7:4] (In ARMv8.1)

Number of VMID bits. Defined values are:

0000 8 bits

0010 16 bits

All other values are reserved.

Bits [7:4] (In ARMv8.0)

Reserved, RES0.

HAFDBS, bits [3:0] (In ARMv8.1)

Hardware updates to Access flag and Dirty state in translation tables. Defined values are:

0000 No hardware update of the Access flag and dirty state is supported in hardware.

0001 Hardware update of the Access flag is supported in hardware.

0010 Hardware update of both the Access flag and dirty state is supported in hardware.

All other values are reserved.

Bits [3:0] (In ARMv8.0)

Reserved, RES0.

Accessing the ID_AA64MMFR1_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ID_AA64MMFR1_EL1	11	000	0000	0111	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ID_AA64MMFR1_EL1	x	x	0	-	RO	n/a	RO
ID_AA64MMFR1_EL1	x	0	1	-	RO	RO	RO
ID_AA64MMFR1_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

B12.2.26 ID_DFR0_EL1, AArch32 Debug Feature Register 0

The ID_DFR0_EL1 characteristics are:

Purpose

Provides top level information about the debug system in AArch32.

Configurations

AArch64 System register ID_DFR0_EL1 is architecturally mapped to AArch32 System register [ID_DFR0](#).

In an AArch64-only implementation, this register is UNKNOWN.

Attributes

ID_DFR0_EL1 is a 32-bit register.

Field descriptions

The ID_DFR0_EL1 bit assignments are:

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0	PerfMon	MProfDbg	MMapTrc	CopTrc	MMapDbg	CopSDBG	CopDbg								

Bits [31:28]

Reserved, RES0.

PerfMon, bits [27:24]

Performance Monitors. Support for System registers-based ARM Performance Monitors Extension, using registers in the coproc == 1111 encoding space, for A and R profile processors. Defined values are:

0000	Performance Monitors Extension system registers not implemented.
0001	Support for Performance Monitors Extension version 1 (PMUv1) System registers.
0010	Support for Performance Monitors Extension version 2 (PMUv2) System registers.
0011	Support for Performance Monitors Extension version 3 (PMUv3) System registers.
0100	Support for Performance Monitors Extension version 3 (PMUv3) System registers, with a 16-bit evtCount field.
1111	IMPLEMENTATION DEFINED form of Performance Monitors System registers supported. PMUv3 not supported.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0011, and 1111.

In ARMv8.1 the permitted values are 0000, 0100, and 1111.

In ARMv7, the value 0000 can mean that PMUv1 is implemented. PMUv1 is not permitted in an ARMv8 implementation.

MProfDbg, bits [23:20]

M Profile Debug. Support for memory-mapped debug model for M profile processors. Defined values are:

0000	Not supported.
0001	Support for M profile Debug architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

MMapTrc, bits [19:16]

Memory Mapped Trace. Support for memory-mapped trace model. Defined values are:

0000 Not supported.

0001 Support for ARM trace architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In the Trace registers, the ETMIDR gives more information about the implementation.

CopTrc, bits [15:12]

Support for System registers-based trace model, using registers in the coproc == 1110 encoding space. Defined values are:

0000 Not supported.

0001 Support for ARM trace architecture, with System registers access.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In the Trace registers, the ETMIDR gives more information about the implementation.

MMapDbg, bits [11:8]

Memory Mapped Debug. Support for v7 memory-mapped debug model, for A and R profile processors.

In ARMv8-A this field is RES0.

The optional memory map defined by ARMv8 is not compatible with ARMv7.

CopSDBG, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = 1110 encoding space, for an A profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-Secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

CopDbg, bits [3:0]

Support for System registers-based debug model, using registers in the coproc == 1110 encoding space, for A and R profile processors. Defined values are:

0000 Not supported.

0010 Support for ARMv6, v6 Debug architecture, with System registers access.

0011 Support for ARMv6, v6.1 Debug architecture, with System registers access.

0100 Support for ARMv7, v7 Debug architecture, with System registers access.

0101 Support for ARMv7, v7.1 Debug architecture, with System registers access.

0110 Support for ARMv8 debug architecture, with System registers access.

0111 Support for ARMv8 debug architecture, with System registers access, and Virtualization Host extensions.

All other values are reserved.

In ARMv8-A the permitted values are 0000, and 0110.

In ARMv8.1 the permitted values are 0000, and 0111.

Accessing the ID_DFR0_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ID_DFR0_EL1	11	000	0000	0001	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ID_DFR0_EL1	x	x	0	-	RO	n/a	RO
ID_DFR0_EL1	x	0	1	-	RO	RO	RO
ID_DFR0_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TID3](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TID3](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

B12.2.27 ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5

The ID_ISAR5_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32.

Configurations

AArch64 System register ID_ISAR5_EL1 is architecturally mapped to AArch32 System register [ID_ISAR5](#).

In an AArch64-only implementation, this register is UNKNOWN.

Attributes

ID_ISAR5_EL1 is a 32-bit register.

Field descriptions

The ID_ISAR5_EL1 bit assignments are:

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0	RDM	RES0	CRC32	SHA2	SHA1	AES	SEVL								

Bits [31:28]

Reserved, RES0.

RDM, bits [27:24] (In ARMv8.1)

VQRDMLAH and VQRDMLSH instructions in AArch32. Defined values are:

0000 No VQRDMLAH and VQRDMLSH instructions implemented.

0001 VQRDMLAH and VQRDMLSH instructions implemented.

All other values are reserved.

In ARMv8.0 the only permitted value is 0000.

In ARMv8.1 the only permitted value is 0001.

Bits [27:24] (In ARMv8.0)

Reserved, RES0.

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates whether CRC32 instructions are implemented in AArch32.

0000 No CRC32 instructions implemented.

0001 CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In ARMv8.1 the only permitted value is 0001.

SHA2, bits [15:12]

Indicates whether SHA2 instructions are implemented in AArch32.

0000 No SHA2 instructions implemented.

0001 SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

SHA1, bits [11:8]

Indicates whether SHA1 instructions are implemented in AArch32.

0000 No SHA1 instructions implemented.

0001 SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

AES, bits [7:4]

Indicates whether AES instructions are implemented in AArch32.

0000 No AES instructions implemented.

0001 AESE, AESD, AESMC, and AESIMC implemented.

0010 As for 0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0010.

SEVL, bits [3:0]

Indicates whether the SEVL instruction is implemented in AArch32.

0000 SEVL is implemented as a NOP.

0001 SEVL is implemented as Send Event Local.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

Accessing the ID_ISAR5_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ID_ISAR5_EL1	11	000	0000	0010	101

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ID_ISAR5_EL1	x	x	0	-	RO	n/a	RO
ID_ISAR5_EL1	x	0	1	-	RO	RO	RO
ID_ISAR5_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547*. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

RETIRED

B12.2.28 ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3

The ID_MMFR3_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32.

Configurations

AArch64 System register ID_MMFR3_EL1 is architecturally mapped to AArch32 System register [ID_MMFR3](#).

In an AArch64-only implementation, this register is UNKNOWN.

Attributes

ID_MMFR3_EL1 is a 32-bit register.

Field descriptions

The ID_MMFR3_EL1 bit assignments are:

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Supersec	CMemSz	CohWalk	PAN	MaintBcst	BPMaint	CMaintSW	CMaintVA								

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported.

Defined values are:

0000 Supersections supported.

1111 Supersections not supported.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 1111.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

0000 4GB, corresponding to a 32-bit physical address range.

0001 64GB, corresponding to a 36-bit physical address range.

0010 1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0001, and 0010.

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the point of unification. Defined values are:

0000 Updates to the translation tables require a clean to the point of unification to ensure visibility by subsequent translation table walks.

0001 Updates to the translation tables do not require a clean to the point of unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

PAN, bits [19:16] (In ARMv8.1)

Privileged Access Never. Indicates support for the PAN bit in [CPSR](#), [SPSR](#), and [DPSR](#) in AArch32. Defined values are:

0000 PAN not supported.

0001 PAN supported.

All other values are reserved.

In ARMv8.1 the only permitted value is 0001.

Bits [19:16] (In ARMv8.0)

Reserved, RES0.

MaintBest, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

0000 Cache, TLB, and branch predictor operations only affect local structures.

0001 Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

0010 Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Defined values are:

0000 None supported.

0001 Supported branch predictor maintenance operations are:

- Invalidate all branch predictors.

0010 As for 0001, and adds:

- Invalidate branch predictors by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance instructions by set/way are:

- Invalidate data cache by set/way.
- Clean data cache by set/way.
- Clean and invalidate data cache by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance operations by VA are:

- Invalidate data cache by VA.
- Clean data cache by VA.
- Clean and invalidate data cache by VA.
- Invalidate instruction cache by VA.
- Invalidate all instruction cache entries.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Accessing the ID_MMFR3_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
ID_MMFR3_EL1	11	000	0000	0001	111

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ID_MMFR3_EL1	x	x	0	-	RO	n/a	RO
ID_MMFR3_EL1	x	0	1	-	RO	RO	RO
ID_MMFR3_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TID3==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

B12.2.29 LORC_EL1, LORegion Control (EL1)

The LORC_EL1 characteristics are:

Purpose

Enables and disables LORegions, and selects the current LORegion descriptor.

Configurations

If no LORegion descriptors are supported by the PE, then this register is RES0.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

LORC_EL1 is a 64-bit register.

Field descriptions

The LORC_EL1 bit assignments are:



Bits [63:10]

Reserved, RES0.

DS, bits [9:2]

Descriptor Select. Selects the current LORegion descriptor accessed by [LORSA_EL1](#), [LOREA_EL1](#), and [LORN_EL1](#).

The number of LORegion descriptors in IMPLEMENTATION DEFINED. The maximum number of LORegion descriptors supported is 256. If the number is less than 256, then bits[63:M+2] are RES0, where M is $\text{Log}_2(\text{Number of LORegion descriptors supported by the implementation})$.

If this field points to an LORegion descriptor that is not supported by an implementation, then the registers [LORN_EL1](#), [LOREA_EL1](#), and [LORSA_EL1](#) are RES0.

Bit [1]

Reserved, RES0.

EN, bit [0]

Enable. Indicates whether LORegions are enabled:

0 Disabled. Memory accesses do not match any LORegions.

1 Enabled. Memory accesses may match a LORegion.

This bit is permitted to be cached in a TLB.

Accessing the LORC_EL1

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
LORC_EL1	11	000	1010	0100	011

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
LORC_EL1	x	x	0	-	-	n/a	-
LORC_EL1	x	0	1	-	RW	RW	RW
LORC_EL1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [SCR_EL3.TLOR](#)==1, Non-secure accesses to this register from EL1 and EL2 are trapped to EL3.

B12.2.30 LOREA_EL1, LORegion End Address (EL1)

The LOREA_EL1 characteristics are:

Purpose

Holds the physical address of the end of the LORegion described in the current LORegion descriptor selected by [LORC_EL1.DS](#).

Configurations

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- [LORC_EL1.DS](#) points to a LORegion that is not supported by the PE.

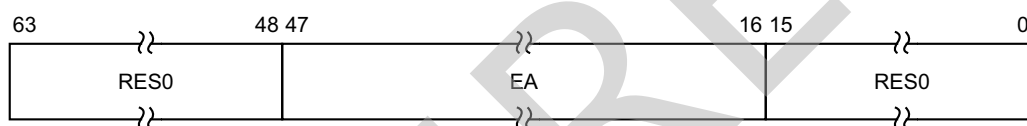
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

LOREA_EL1 is a 64-bit register.

Field descriptions

The LOREA_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:48]

Reserved, RES0.

EA, bits [47:16]

Bits [47:16] of the end physical address of an LORegion described in the current LORegion descriptor selected by [LORC_EL1.DS](#). Bits[15:0] of this address are defined to be 0xFFFF.

Bits [15:0]

Reserved, RES0.

Accessing the LOREA_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
LOREA_EL1	11	000	1010	0100	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
LOREA_EL1	x	x	0	-	-	n/a	-
LOREA_EL1	x	0	1	-	RW	RW	RW
LOREA_EL1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [SCR_EL3.TLOR](#)==1, Non-secure accesses to this register from EL1 and EL2 are trapped to EL3.

B12.2.31 LORID_EL1, LORegionID (EL1)

The LORID_EL1 characteristics are:

Purpose

Indicates the number of LORegions and LORegion descriptors supported by the PE.

Configurations

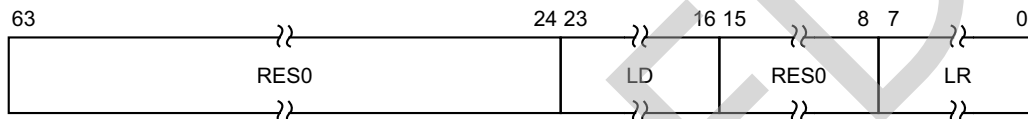
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

LORID_EL1 is a 64-bit register.

Field descriptions

The LORID_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

LD, bits [23:16]

Number of LORegion descriptors supported by the PE. This is an 8-bit binary number.

Bits [15:8]

Reserved, RES0.

LR, bits [7:0]

Number of LORegions supported by the PE. This is an 8-bit binary number.

Note

If LORID_EL1 indicates that no LORegions are implemented, then LoadLOAcquire and StoreLORelease will behave as LoadAcquire and StoreRelease.

Accessing the LORID_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
LORID_EL1	11	000	1010	0100	111

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
LORID_EL1	x	x	0	-	RO	n/a	RO
LORID_EL1	x	0	1	-	RO	RO	RO
LORID_EL1	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If [SCR_EL3.TLOR](#)==1, accesses to this register from EL1 and EL2 are trapped to EL3.

B12.2.32 LORN_EL1, LORegion Number (EL1)

The LORN_EL1 characteristics are:

Purpose

Holds the number of the LORegion described in the current LORegion descriptor selected by [LORC_EL1.DS](#).

Configurations

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- [LORC_EL1.DS](#) points to a LORegion that is not supported by the PE.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

LORN_EL1 is a 64-bit register.

Field descriptions

The LORN_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:8]

Reserved, RES0.

Num, bits [7:0]

Number of the LORegion described in the current LORegion descriptor selected by [LORC_EL1.DS](#).

The maximum number of LORegions supported by the PE is 256. If the maximum number is less than 256, then bits[8:N] are RES0, where N is ($\text{Log}_2(\text{Number of LORegions supported by the PE})$).

If this field points to a LORegion that is not supported by the PE, then the current LORegion descriptor does not match any LORegion.

Accessing the LORN_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
LORN_EL1	11	000	1010	0100	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
LORN_EL1	x	x	0	-	-	n/a	-
LORN_EL1	x	0	1	-	RW	RW	RW
LORN_EL1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [SCR_EL3.TLOR](#)==1, Non-secure accesses to this register from EL1 and EL2 are trapped to EL3.

B12.2.33 LORSA_EL1, LORegion Start Address (EL1)

The LORSA_EL1 characteristics are:

Purpose

Indicates whether the current LORegion descriptor selected by [LORC_EL1.DS](#) is enabled, and holds the physical address of the start of the LORegion.

Configurations

This register is RES0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- [LORC_EL1.DS](#) points to a LORegion that is not supported by the PE.

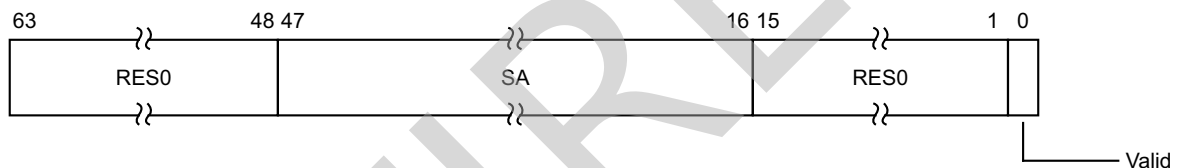
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

LORSA_EL1 is a 64-bit register.

Field descriptions

The LORSA_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

Bits [63:48]

Reserved, RES0.

SA, bits [47:16]

Bits [47:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by [LORC_EL1.DS](#). Bits[15:0] of this address are defined to be 0x0000.

Bits [15:1]

Reserved, RES0.

Valid, bit [0]

Indicates whether the current LORegion Descriptor is enabled.

- | | |
|---|----------|
| 0 | Disabled |
| 1 | Enabled |

Accessing the LORSA_EL1

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
LORSA_EL1	11	000	1010	0100	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
LORSA_EL1	x	x	0	-	-	n/a	-
LORSA_EL1	x	0	1	-	RW	RW	RW
LORSA_EL1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TLOR](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [SCR_EL3.TLOR](#)==1, Non-secure accesses to this register from EL1 and EL2 are trapped to EL3.

B12.2.34 MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIdx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

Configurations

AArch64 System register MAIR_EL1[31:0] is architecturally mapped to AArch32 System register PRRR when TTBCR.EAE==0.

AArch64 System register MAIR_EL1[31:0] is architecturally mapped to AArch32 System register MAIR0 when TTBCR.EAE==1.

AArch64 System register MAIR_EL1[63:32] is architecturally mapped to AArch32 System register NMRR when TTBCR.EAE==0.

AArch64 System register MAIR_EL1[63:32] is architecturally mapped to AArch32 System register MAIR1 when TTBCR.EAE==1.

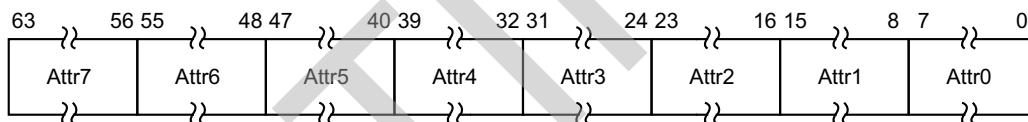
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

MAIR_EL1 is a 64-bit register.

Field descriptions

The MAIR_EL1 bit assignments are:



Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIdx[2:0] entry in a Long descriptor format translation table entry, where AttrIdx[2:0] gives the value of <n> in Attr<n>.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
0000	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
00RW, RW not 00	Normal Memory, Outer Write-through transient
0100	Normal Memory, Outer Non-cacheable
01RW, RW not 00	Normal Memory, Outer Write-back transient
10RW	Normal Memory, Outer Write-through non-transient
11RW	Normal Memory, Outer Write-back non-transient

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0000	Meaning when Attr<n>[7:4] is not 0000
0000	Device-nGnRnE memory	UNPREDICTABLE
00RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-through transient
0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
01RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-back transient
1000	Device-nGRE memory	Normal Memory, Inner Write-through non-transient (RW=00)
10RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-through non-transient
1100	Device-GRE memory	Normal Memory, Inner Write-back non-transient (RW=00)
11RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-back non-transient

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0	Do not allocate
1	Allocate

Accessing the MAIR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
MAIR_EL1	11	000	1010	0010	000
MAIR_EL12	11	101	1010	0010	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
MAIR_EL1	x	x	0	-	RW	n/a	RW
MAIR_EL1	0	0	1	-	RW	RW	RW
MAIR_EL1	0	1	1	-	n/a	RW	RW

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
MAIR_EL1	1	0	1	-	RW	MAIR_EL2	RW
MAIR_EL1	1	1	1	-	n/a	MAIR_EL2	RW
MAIR_EL12	x	x	0	-	-	n/a	-
MAIR_EL12	0	0	1	-	-	-	-
MAIR_EL12	0	1	1	-	n/a	-	-
MAIR_EL12	1	0	1	-	-	RW	RW
MAIR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic MAIR_EL1 or MAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.35 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIdx values in a Long-descriptor format translation table entry for stage 1 translations at EL2.

Configurations

AArch64 System register MAIR_EL2[31:0] is architecturally mapped to AArch32 System register HMAIR0.

AArch64 System register MAIR_EL2[63:32] is architecturally mapped to AArch32 System register HMAIR1.

If EL2 is not implemented, this register is RES0 from EL3.

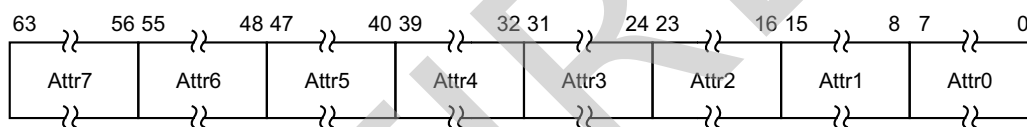
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

MAIR_EL2 is a 64-bit register.

Field descriptions

The MAIR_EL2 bit assignments are:



Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIdx[2:0] entry in a Long descriptor format translation table entry, where AttrIdx[2:0] gives the value of <n> in Attr<n>.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
0000	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
00RW, RW not 00	Normal Memory, Outer Write-through transient
0100	Normal Memory, Outer Non-cacheable
01RW, RW not 00	Normal Memory, Outer Write-back transient
10RW	Normal Memory, Outer Write-through non-transient
11RW	Normal Memory, Outer Write-back non-transient

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0000	Meaning when Attr<n>[7:4] is not 0000
0000	Device-nGnRnE memory	UNPREDICTABLE
00RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-through transient
0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
01RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-back transient
1000	Device-nGRE memory	Normal Memory, Inner Write-through non-transient (RW=00)
10RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-through non-transient
1100	Device-GRE memory	Normal Memory, Inner Write-back non-transient (RW=00)
11RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-back non-transient

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0	Do not allocate
1	Allocate

Accessing the MAIR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
MAIR_EL2	11	100	1010	0010	000
MAIR_EL1	11	000	1010	0010	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
MAIR_EL2	x	x	0	-	-	n/a	RW
MAIR_EL2	0	0	1	-	-	RW	RW
MAIR_EL2	0	1	1	-	n/a	RW	RW

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
MAIR_EL2	1	0	1	-	-	RW	RW
MAIR_EL2	1	1	1	-	n/a	RW	RW
MAIR_EL1	x	x	0	-	MAIR_EL1	n/a	MAIR_EL1
MAIR_EL1	0	0	1	-	MAIR_EL1	MAIR_EL1	MAIR_EL1
MAIR_EL1	0	1	1	-	n/a	MAIR_EL1	MAIR_EL1
MAIR_EL1	1	0	1	-	MAIR_EL1	RW	MAIR_EL1
MAIR_EL1	1	1	1	-	n/a	RW	MAIR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic MAIR_EL2 or MAIR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.36 PAN, Privileged Access Never

The PAN characteristics are:

Purpose

Allows access to the Privileged Access Never bit.

Configurations

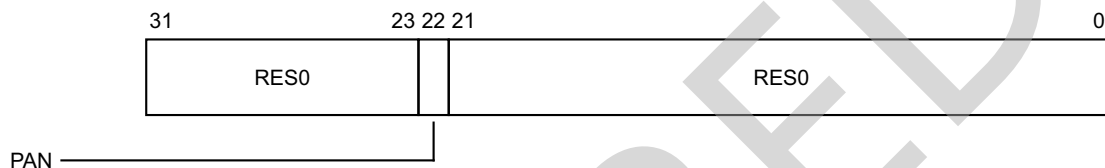
There are no configuration notes.

Attributes

PAN is a 32-bit register.

Field descriptions

The PAN bit assignments are:



Bits [31:23]

Reserved, RES0.

PAN, bit [22]

Privileged Access Never. Defined values are:

- 0 The translation system is the same as ARMv8.0.
- 1 Disables privileged read and write accesses.

On taking an exception from the current mode or Exception level, to EL1, EL2 or EL3:

- If the target is EL1, this bit is copied to [SPSR_EL1](#).
- If the target is EL2, this bit is copied to [SPSR_EL2](#).
- If the target is EL3, this bit is copied to [SPSR_EL2](#).

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the [SCTLR_EL1.SPAN](#) bit is 0, this bit is set to 1.
- When the target of the exception is EL2, [HCR_EL2](#).{E2H, TGE} is {1, 1}, and the value of the [SCTLR_EL2.SPAN](#) bit is 0, this bit is set to 1.

Bits [21:0]

Reserved, RES0.

Accessing the PAN

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
PAN	11	000	0100	0010	011

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
PAN	x	x	0	-	RW	n/a	RW
PAN	x	0	1	-	RW	RW	RW
PAN	x	1	1	-	n/a	RW	RW

B12.2.37 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

Purpose

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0 and EL1, either Secure or Non-secure.
- The Execution state at lower Exception levels.
- Whether IRQ, FIQ, and External Abort interrupts are taken to EL3.

Configurations

AArch64 System register SCR_EL3 can be mapped to AArch32 System register SCR, but this is not architecturally mandated.

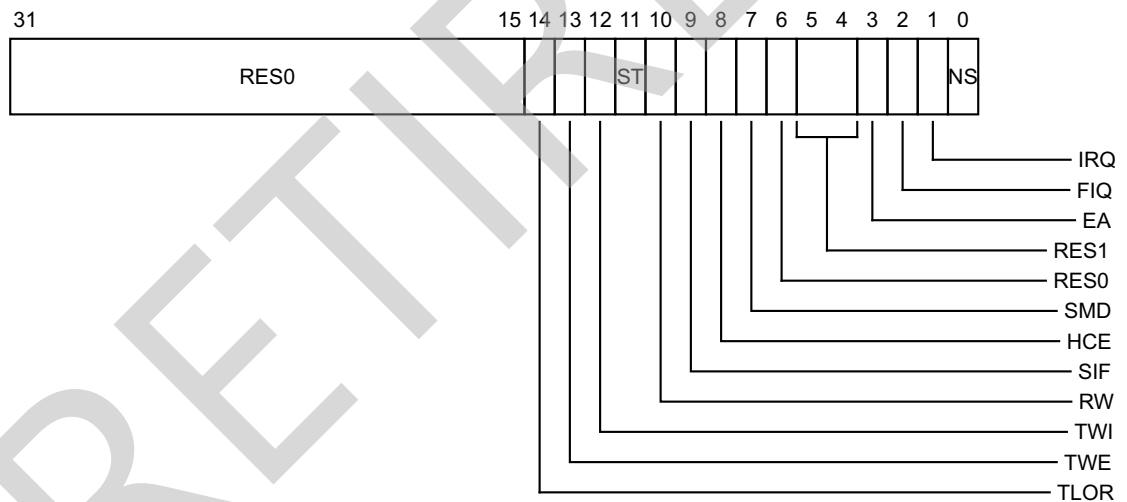
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

SCR_EL3 is a 32-bit register.

Field descriptions

The SCR_EL3 bit assignments are:



Bits [31:15]

Reserved, RES0.

TLOR, bit [14] (In ARMv8.1)

Trap LOR registers. Traps accesses to the [LORSA_EL1](#), [LOREA_EL1](#), [LORN_EL1](#), [LORC_EL1](#), and [LORID_EL1](#) registers from EL1 and EL2 to EL3, unless the access has been trapped to EL2.

0 This control has no effect on EL1 and EL2 accesses to the LOR registers.

1 EL1 and EL2 accesses to the LOR registers that are not UNDEFINED are trapped to EL3, unless the access has been trapped to EL2 as a result of HCR_EL2.TLOR being set to 1.

When [HCR_EL2.TGE](#) is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Bit [14] (In ARMv8.0)

Reserved, RES0.

TWE, bit [13]

Traps EL2, EL1, and EL0 execution of WFE instructions to EL3, from both Security states and both Execution states.

- | | |
|---|---|
| 0 | EL2, EL1, and EL0 execution of WFE instructions is not trapped to EL3. |
| 1 | Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state. |

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

———— Note ————

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

TWI, bit [12]

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from both Security states and both Execution states.

- | | |
|---|---|
| 0 | EL2, EL1, and EL0 execution of WFI instructions is not trapped to EL3. |
| 1 | Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state. |

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

———— Note ————

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

ST, bit [11]

Traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3, from AArch64 state only.

- | | |
|---|--|
| 0 | Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are trapped to EL3. |
| 1 | Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are not trapped to EL3. |

RW, bit [10]

Execution state control for lower Exception levels.

- | | |
|---|--|
| 0 | Lower levels are all AArch32. |
| 1 | <p>The next lower level is AArch64.</p> <p>If EL2 is present:</p> <ul style="list-style-type: none"> • EL2 is AArch64. • EL2 controls EL1 and EL0 behaviors. <p>If EL2 is not present:</p> <ul style="list-style-type: none"> • EL1 is AArch64. • EL0 is determined by the Execution state described in the current process state when executing at EL0. |

If all lower exception levels cannot use AArch32 then this bit is RAO/WI.

This bit is permitted to be cached in a TLB.

SIF, bit [9]

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from Non-secure memory. The possible values for this bit are:

- 0 Secure state instruction fetches from Non-secure memory are permitted.
- 1 Secure state instruction fetches from Non-secure memory are not permitted.

This bit is permitted to be cached in a TLB.

HCE, bit [8]

Hypervisor Call instruction enable. Enables HVC instructions at EL3, EL2, and Non-secure EL1, in both Execution states.

- 0 HVC instructions are UNDEFINED at EL3, EL2, and Non-secure EL1, and any resulting exception is taken from the current Exception level to the current Exception level.
- 1 HVC instructions are enabled at EL1 and above.

————— Note —————

HVC instructions are always UNDEFINED at EL0.

If EL2 is not implemented, this bit is RES0.

SMD, bit [7]

Secure Monitor Call disable. Disables SMC instructions at EL1 and above, from both Security states and both Execution states.

- 0 SMC instructions are enabled at EL1 and above.
- 1 SMC instructions are UNDEFINED at EL1 and above.

————— Note —————

SMC instructions are always UNDEFINED at EL0.

Bit [6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError Interrupt Routing.

- 0 When executing at Exception levels below EL3, External Aborts and SError Interrupts are not taken to EL3.
In addition, when executing at EL3:
 - SError Interrupts are not taken.
 - External Aborts are taken to EL3.
- 1 When executing at any Exception level, External Aborts and SError Interrupts are taken to EL3.

For more information, see [Asynchronous exception routing on page B12-550](#).

FIQ, bit [2]

Physical FIQ Routing.

- 0 When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3.

When executing at EL3, physical FIQ interrupts are not taken.

- 1 When executing at any Exception level, physical FIQ interrupts are taken to EL3.
For more information, see [Asynchronous exception routing](#) on page B12-550.

IRQ, bit [1]

Physical IRQ Routing.

- 0 When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3.
When executing at EL3, physical IRQ interrupts are not taken.
- 1 When executing at any Exception level, physical IRQ interrupts are taken to EL3.
For more information, see [Asynchronous exception routing](#) on page B12-550.

NS, bit [0]

Non-secure bit.

- 0 Indicates that EL0 and EL1 are in Secure state, and so memory accesses from those Exception levels can access Secure memory.
When executing at EL3:
- The AT S1E2R, AT S1E2W, TLBI VAE2, TLBI VALE2, TLBI VAE2IS, TLBI VALE2IS, TLBI ALLE2, and TLBI ALLE2IS System instructions are UNDEFINED.
 - Each AT S1E2* System instruction executes as the corresponding AT S1E* instruction. For example, [AT S1E2R](#) executes as [AT S1ER](#).
 - Each of the TLBI IPAS2E1, TLBI IPAS2E1IS, TLBI IPAS2LE1, and TLBI IPAS2LE1IS System instructions executes as a NOP.
 - A TLBI VMALLS12E1 System instruction executes as [TLBI VMALLE1](#), and a TLBI VMALLS12E1IS System instruction executes as [TLBI VMALLE1IS](#).
- 1 Indicates that EL0 and EL1 are in Non-secure state, and so memory accesses from those Exception levels cannot access Secure memory.

Note

EL2 is not supported in the Secure state. When $SCR_EL3.NS == 0$, it is not possible to enter EL2, and the EL2 state has no effect on execution. See “Virtualization” in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

Accessing the SCR_EL3

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SCR_EL3	11	110	0001	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SCR_EL3	x	x	0	-	-	n/a	RW
SCR_EL3	0	0	1	-	-	-	RW
SCR_EL3	0	1	1	-	n/a	-	RW
SCR_EL3	1	0	1	-	-	-	RW
SCR_EL3	1	1	1	-	n/a	-	RW

B12.2.38 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL1 and EL0.

Configurations

AArch64 System register SCTLR_EL1 is architecturally mapped to AArch32 System register [SCTLR](#).

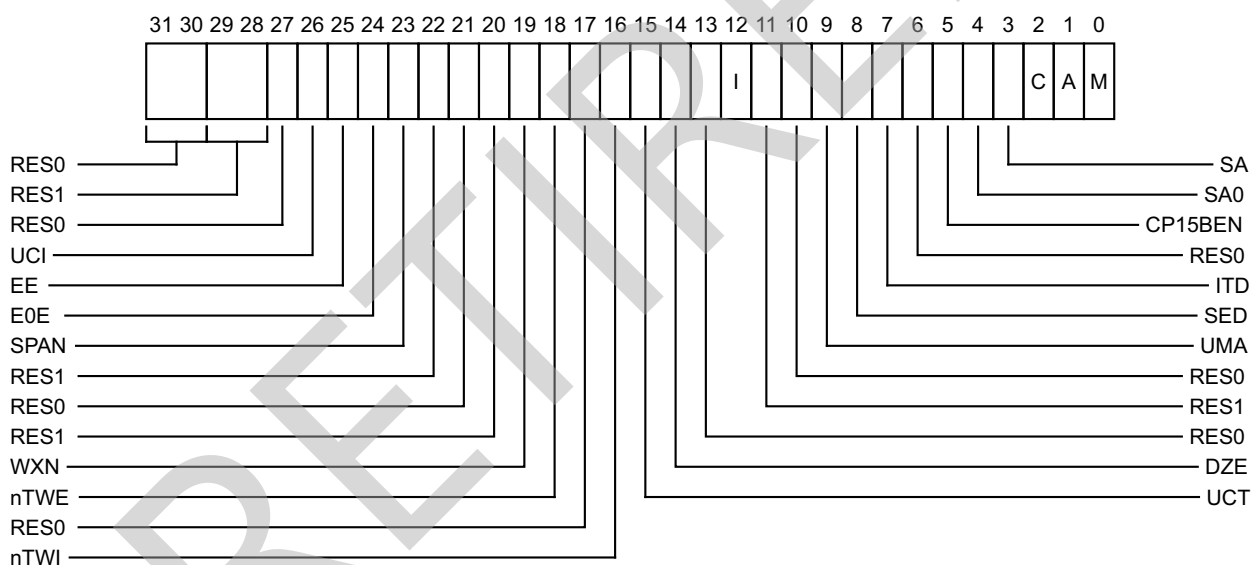
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL1 using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

Attributes

SCTLR_EL1 is a 32-bit register.

Field descriptions

The SCTLR_EL1 bit assignments are:



Bits [31:30]

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

Bit [27]

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions to EL1, from AArch64 state only.

0 Any attempt to execute a DC CVAU, DC CIVAC, DC CVAC, or IC IVAU instruction at EL0 using AArch64 is trapped to EL1.

1 Does not cause any instruction to be trapped.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

EE, bit [25]

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

The possible values of this bit are:

- 0 Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.
- 1 Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on the PE.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

The possible values of this bit are:

- 0 Explicit data accesses at EL0 are little-endian.
- 1 Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0 then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0 then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR and STTRH instructions executed at EL1.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

SPAN, bit [23] (In ARMv8.1)

Set Privileged Access Never, on taking an exception to EL1.

- 0 PSTATE.PAN is set to 1 on taking an exception to EL1.
- 1 The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [23] (In ARMv8.0)

Reserved, RES1.

Bit [22]

Reserved, RES1.

Bit [21]

Reserved, RES0.

Bit [20]

Reserved, RES1.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

- 0 This control has no effect on memory access permissions.
- 1 Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

The WXN bit is permitted to be cached in a TLB.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on the PE.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, from both Execution states.

- 0 Any attempt to execute a WFE instruction at EL0 is trapped to EL1, if the instruction would otherwise have caused the PE to enter a low-power state.
- 1 Does not cause any instruction to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL1, from both Execution states.

- 0 Any attempt to execute a WFI instruction at EL0 is trapped EL1, if the instruction would otherwise have caused the PE to enter a low-power state.
- 1 Does not cause any instruction to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, from AArch64 state only.

0 Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL1.

1 Does not cause any instruction to be trapped.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL1, from AArch64 state only.

0 Any attempt to execute a DC ZVA instruction at EL0 using AArch64 is trapped to EL1. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that DC ZVA instructions are not supported.

1 Does not cause any instruction to be trapped.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL0 and EL1:

0 All instruction access to Normal memory from EL0 and EL1 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLRL_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

1 This control has no effect on the Cacheability of instruction access to Normal memory from EL0 and EL1.

If the value of SCTLRL_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the [HCR_EL2](#).DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLRL_EL1.I bit.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on the PE.

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [11]

Reserved, RES1.

Bit [10]

Reserved, RES0.

UMA, bit [9]

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE. {D, A, I, F} masks to EL1, from AArch64 state only.

0 Any attempt at EL0 using AArch64 to execute an MRS, MSR(register), or MSR(immediate) instruction that accesses the DAIF is trapped to EL1.

1 Does not cause any instruction to be trapped.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

0 SETEND instruction execution is enabled at EL0 using AArch32.

1 SETEND instructions are UNDEFINED at EL0 using AArch32.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

If EL0 cannot use AArch32, this bit is RES1.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

0 All IT instruction functionality is enabled at EL0 using AArch32.

1 Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with `hw1[3:0] != 1000`.
- All encodings of the subsequent instruction with the following values for `hw1`:

11xxxxxxxxxxxx

All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

1011xxxxxxxxxxxx

All instructions in [Miscellaneous 16-bit instructions on page C4-573](#).

1010xxxxxxxxxxxx

ADD Rd, PC, #imm

01001xxxxxxxxxxx

LDR Rd, [PC, #imm]

0100x1xxx1111xxx

ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

010001xx1xxxx111

ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If EL0 cannot use AArch32, this bit is RES1.

ITD is optional, but if it is implemented in the [SCTLR](#) then it must also be implemented in the [SCTLR_EL1](#). If it is not implemented then this bit is RAZ/WI.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bit [6]

Reserved, RES0.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==1111) encoding space from EL0:

- | | |
|---|--|
| 0 | EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED. |
| 1 | EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled. |

If EL0 cannot use AArch32, this bit is RES0.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

In ARMv8.1, if HCR_EL2.{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

SA0, bit [4]

Stack Alignment Check Enable for EL0. When set, use of the stack pointer as the base address in a load/store instruction at EL0 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

In ARMv8.1, if HCR_EL2.{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

SA, bit [3]

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at EL1 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

In ARMv8.1, if HCR_EL2.{E2H, TGE} is set to {1, 1}, this bit has no effect on the PE.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

C, bit [2]

Cacheability control, for data accesses.

- | | |
|---|--|
| 0 | All data access to Normal memory from EL0 and EL1, and all Normal memory accesses to the EL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache. |
| 1 | This control has no effect on the Cacheability of: <ul style="list-style-type: none"> • Data access to Normal memory from EL0 and EL1. • Normal memory accesses to the EL1&0 stage 1 translation tables. |

When the value of the HCR_EL2.DC bit is 1, the PE ignores SCLTR.C. This means that Non-secure EL0 and Non-secure EL1 data accesses to Normal memory are Cacheable.

In ARMv8.1, if HCR_EL2.{E2H, TGE} is set to {1, 1}, this bit has no effect on the PE.

When this register has an architecturally-defined reset value, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0:

- | | |
|---|---|
| 0 | Alignment fault checking disabled when executing at EL1 or EL0. |
|---|---|

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

- 1 Alignment fault checking enabled when executing at EL1 or EL0.
All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on execution at EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

M, bit [0]

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

- 0 EL1 and EL0 stage 1 address translation disabled.
See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.
- 1 EL1 and EL0 stage 1 address translation enabled.

If the value of [HCR_EL2](#).{DC, TGE} is not {0, 0} then in Non-secure state the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.

In ARMv8.1, if [HCR_EL2](#).{E2H, TGE} is set to {1, 1}, this bit has no effect on the PE.

When this register has an architecturally-defined reset value, this field resets to 0.

Accessing the SCTLR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SCTLR_EL1	11	000	0001	0000	000
SCTLR_EL12	11	101	0001	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SCTLR_EL1	x	x	0	-	RW	n/a	RW
SCTLR_EL1	0	0	1	-	RW	RW	RW
SCTLR_EL1	0	1	1	-	n/a	RW	RW

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SCTLR_EL1	1	0	1	-	RW	SCTLR_EL2	RW
SCTLR_EL1	1	1	1	-	n/a	SCTLR_EL2	RW
SCTLR_EL12	x	x	0	-	-	n/a	-
SCTLR_EL12	0	0	1	-	-	-	-
SCTLR_EL12	0	1	1	-	n/a	-	-
SCTLR_EL12	1	0	1	-	-	RW	RW
SCTLR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic SCTLR_EL1 or SCTLR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.39 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL2.

In ARMv8.1, when the value of [HCR_EL2](#).{E2H, TGE} is {1, 1}, these controls apply also to execution at EL0.

Configurations

AArch64 System register SCTLR_EL2 is architecturally mapped to AArch32 System register HSCTLR.

If EL2 is not implemented, this register is RES0 from EL3.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

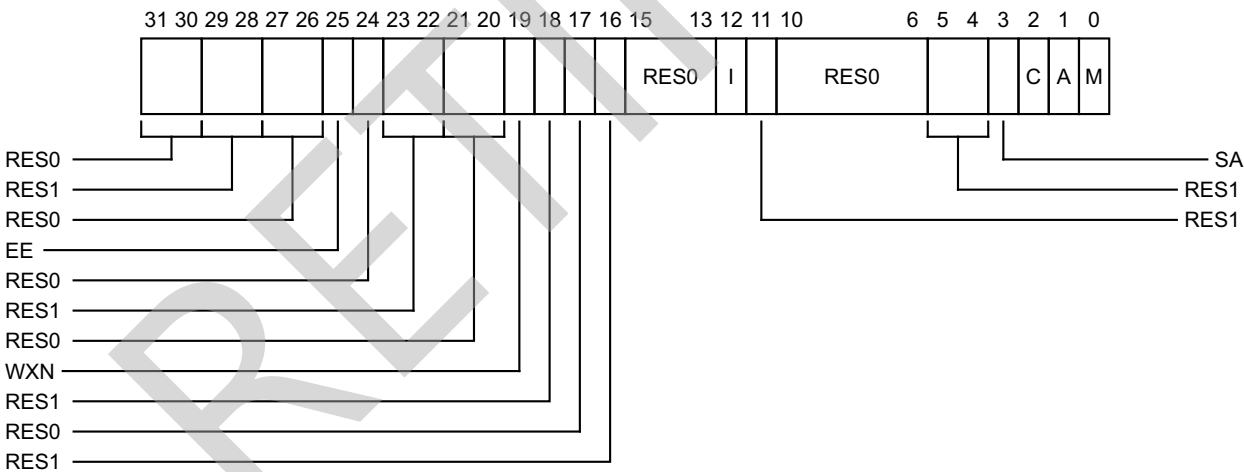
Attributes

SCTLR_EL2 is a 32-bit register.

Field descriptions

The SCTLR_EL2 bit assignments are:

When *HCR_EL2*.{E2H, TGE} != {1, 1}:



This format applies in all ARMv8.0 implementations, and in ARMv8.1 in Secure state.

Bits [31:30]

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the EL1&0 translation regime.

The possible values of this bit are:

- 0 Explicit data accesses at EL2, stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.
- 1 Explicit data accesses at EL2, stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

Bits [21:20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

- 0 This control has no effect on memory access permissions.
- 1 Any region that is writable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Bits [15:13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2:

- 0 All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTL_R_EL2.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

- 1 This control has no effect on the Cacheability of instruction access to Normal memory from EL2.

If the value of SCTL_R_EL2.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0 or EL3 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [11]

Reserved, RES1.

Bits [10:6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at EL2 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

C, bit [2]

Cacheability control, for data accesses.

- 0 All data access to Normal memory from EL2, and all Normal memory accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.

- 1 This control has no effect on the Cacheability of:
- Data access to Normal memory from EL2.
 - Normal memory accesses to the EL2 translation tables.

This bit has no effect on the EL1&0 or EL3 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

- 0 Alignment fault checking disabled when executing at EL2.
Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

- 1 Alignment fault checking enabled when executing at EL2.
All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

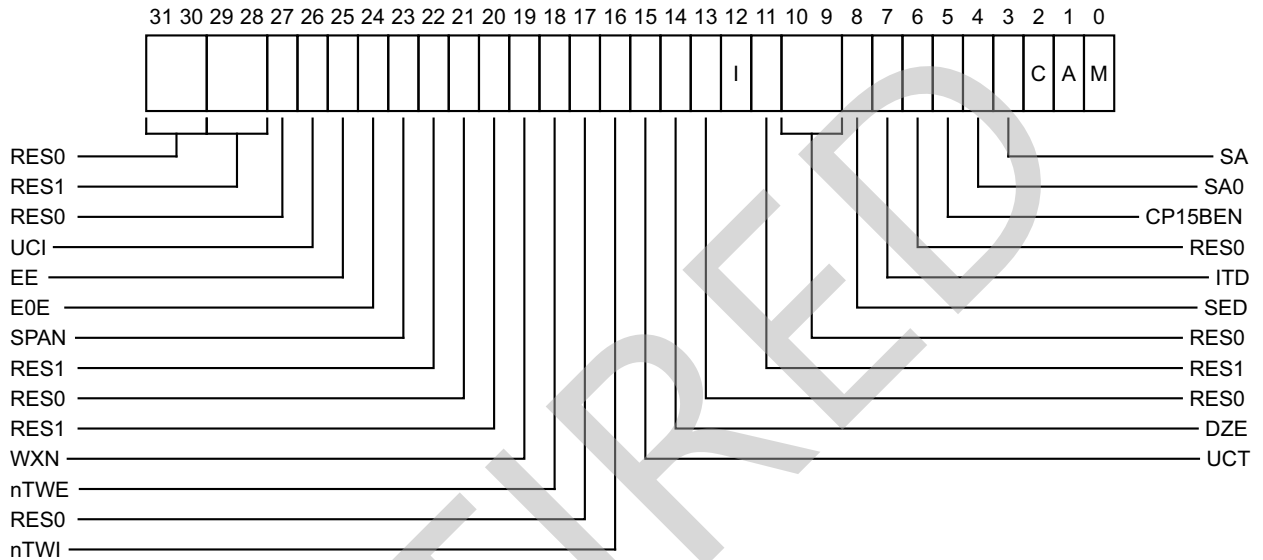
M, bit [0]

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:

- 0 EL2 stage 1 address translation disabled.
See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.
- 1 EL2 stage 1 address translation enabled.

When this register has an architecturally-defined reset value, this field resets to 0.

When $HCR_EL2.\{E2H, TGE\} == \{1, 1\}$:



This format applies in ARMv8.1 in Non-secure state only when $HCR_EL2.\{E2H, TGE\} == \{1, 1\}$.

Bits [31:30]

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

Bit [27]

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions to EL2, from AArch64 state only.

- 0 Any attempt to execute a DC CVAU, DC CIVAC, DC CVAC, or IC IVAU instruction at EL0 using AArch64 is trapped to EL2.
- 1 Does not cause any instruction to be trapped.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

EE, bit [25]

Endianness of data accesses at EL2, and stage 1 translation table walks in the EL2&0 translation regime.

The possible values of this bit are:

- 0 Explicit data accesses at EL2, and stage 1 translation table walks in the EL2&0 translation regime are little-endian.

- 1 Explicit data accesses at EL2, and stage 1 translation table walks in the EL2&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

The possible values of this bit are:

- 0 Explicit data accesses at EL0 are little-endian.

- 1 Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0 then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0 then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR and STTRH instructions executed at EL1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

SPAN, bit [23]

Set Privileged Access Never, on taking an exception to EL2.

- 0 PSTATE.PAN is set to 1 on taking an exception to EL2.

- 1 The value of PSTATE.PAN is left unchanged on taking an exception to EL2.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [22]

Reserved, RES1.

Bit [21]

Reserved, RES0.

Bit [20]

Reserved, RES1.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

- 0 This control has no effect on memory access permissions.

- 1 Any region that is writable in the EL2&0 translation regime is forced to XN for accesses from software executing at EL2 or EL0.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL2, from both Execution states.

- 0 Any attempt to execute a WFE instruction at EL0 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.
- 1 Does not cause any instruction to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

———— Note ————

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL2, from both Execution states.

- 0 Any attempt to execute a WFI instruction at EL0 is trapped EL2, if the instruction would otherwise have caused the PE to enter a low-power state.
- 1 Does not cause any instruction to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

———— Note ————

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL2, from AArch64 state only.

- 0 Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL2.
- 1 Does not cause any instruction to be trapped.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL2, from AArch64 state only.

- 0 Any attempt to execute a DC ZVA instruction at EL0 using AArch64 is trapped to EL2. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that DC ZVA instructions are not supported.
- 1 Does not cause any instruction to be trapped.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2 and EL0:

- 0 All instruction access to Normal memory from EL2 and EL0 are Non-cacheable for all levels of instruction and unified cache.
If the value of SCTL_R_EL2.M is 0, instruction accesses from stage 1 of the EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
- 1 This control has no effect on the Cacheability of instruction access to Normal memory from EL2 and EL0.
If the value of SCTL_R_EL2.M is 0, instruction accesses from stage 1 of the EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL3 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

- 0 SETEND instruction execution is enabled at EL0 using AArch32.
- 1 SETEND instructions are UNDEFINED at EL0 using AArch32.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

If EL0 cannot use AArch32, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

- 0 All IT instruction functionality is enabled at EL0 using AArch32.
- 1 Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED:
- All encodings of the IT instruction with hw1[3:0] != 1000.
 - All encodings of the subsequent instruction with the following values for hw1:

11xxxxxxxxxxxxx
All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

1011xxxxxxxxxxxx
All instructions in [Miscellaneous 16-bit instructions on page C4-573](#).

10100xxxxxxxxxxx
ADD Rd, PC, #imm

01001xxxxxxxxxxx
LDR Rd, [PC, #imm]

0100x1xxx1111xxx

ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

010001xx1xxxx111

ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If EL0 cannot use AArch32, this bit is RES1.

ITD is optional, but if it is implemented in the [SCTLR](#) then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bit [6]

Reserved, RES0.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==1111) encoding space from EL0:

- | | |
|---|--|
| 0 | EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED. |
| 1 | EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled. |

If EL0 cannot use AArch32, this bit is RES0.

CP15BEN is optional, but if it is implemented in the [SCTLR](#) then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

SA0, bit [4]

Stack Alignment Check Enable for EL0. When set, use of the stack pointer as the base address in a load/store instruction at EL0 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

SA, bit [3]

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at EL2 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

C, bit [2]

Cacheability control, for data accesses.

- 0 All data access to Normal memory from EL2 and EL0, and all Normal memory accesses to the EL2&0 translation tables, are Non-cacheable for all levels of data and unified cache.
- 1 This control has no effect on the Cacheability of:
 - Data access to Normal memory from EL2 and EL0.
 - Normal memory accesses to the EL2&0 translation tables.

This bit has no effect on the EL3 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2 and EL0:

- 0 Alignment fault checking disabled when executing at EL2 and EL0.
Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
- 1 Alignment fault checking enabled when executing at EL2 and EL0.
All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

M, bit [0]

MMU enable for EL2&0 stage 1 address translation. Possible values of this bit are:

- 0 EL2&0 stage 1 address translation disabled.
See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.
- 1 EL2&1 stage 1 address translation enabled.

When this register has an architecturally-defined reset value, this field resets to 0.

Accessing the SCTLR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SCTLR_EL2	11	100	0001	0000	000
SCTLR_EL1	11	000	0001	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SCTLR_EL2	x	x	0	-	-	n/a	RW
SCTLR_EL2	0	0	1	-	-	RW	RW
SCTLR_EL2	0	1	1	-	n/a	RW	RW
SCTLR_EL2	1	0	1	-	-	RW	RW
SCTLR_EL2	1	1	1	-	n/a	RW	RW
SCTLR_EL1	x	x	0	-	SCTLR_EL1	n/a	SCTLR_EL1
SCTLR_EL1	0	0	1	-	SCTLR_EL1	SCTLR_EL1	SCTLR_EL1
SCTLR_EL1	0	1	1	-	n/a	SCTLR_EL1	SCTLR_EL1
SCTLR_EL1	1	0	1	-	SCTLR_EL1	RW	SCTLR_EL1
SCTLR_EL1	1	1	1	-	n/a	RW	SCTLR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic [SCTLR_EL2](#) or [SCTLR_EL1](#) are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.40 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configurations

AArch64 System register SPSR_abt is architecturally mapped to AArch32 System register [SPSR_abt](#).

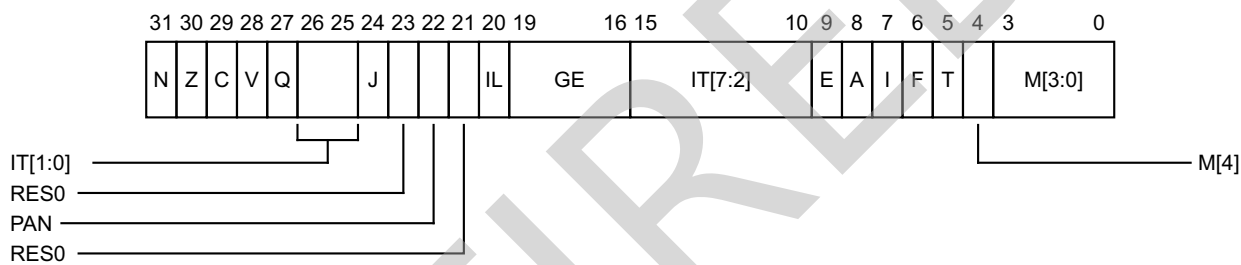
If EL1 does not support execution in AArch32, this register is RES0.

Attributes

SPSR_abt is a 32-bit register.

Field descriptions

The SPSR_abt bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Abort mode, and copied to [CPSR.N](#) on executing an exception return operation in Abort mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Abort mode, and copied to [CPSR.Z](#) on executing an exception return operation in Abort mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Abort mode, and copied to [CPSR.C](#) on executing an exception return operation in Abort mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Abort mode, and copied to [CPSR.V](#) on executing an exception return operation in Abort mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to Abort mode, and copied to [CPSR.PAN](#) on executing an exception return operation in Abort mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved.

Accessing the SPSR_abt

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_abt	11	100	0100	0011	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_abt	x	x	0	-	-	n/a	RW
SPSR_abt	x	0	1	-	-	RW	RW
SPSR_abt	x	1	1	-	n/a	RW	RW

B12.2.41 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL1.

Configurations

AArch64 System register SPSR_EL1 is architecturally mapped to AArch32 System register [SPSR_svc](#).

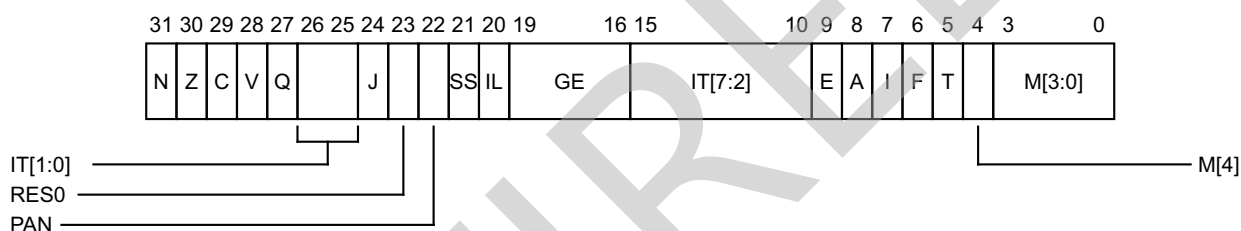
Attributes

SPSR_EL1 is a 32-bit register.

Field descriptions

The SPSR_EL1 bit assignments are:

When exception taken from AArch32:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Supervisor mode, and copied to [CPSR.N](#) on executing an exception return operation in Supervisor mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Supervisor mode, and copied to [CPSR.Z](#) on executing an exception return operation in Supervisor mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Supervisor mode, and copied to [CPSR.C](#) on executing an exception return operation in Supervisor mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Supervisor mode, and copied to [CPSR.V](#) on executing an exception return operation in Supervisor mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of **CPSR.PAN** on taking an exception to Supervisor mode, and copied to **CPSR.PAN** on executing an exception return operation in Supervisor mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of **PSTATE.SS** immediately before the exception was taken.

IL, bit [20]

Illegal Execution state bit. Shows the value of **PSTATE.IL** immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the **SCTLR.EE** bit is defined by a configuration input signal, that value also applies to the **CPSR.E** bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

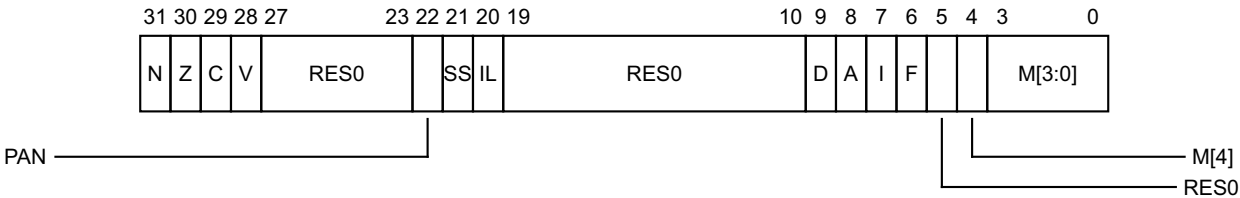
M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0111	Abort
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page B12-558.

When exception taken from AArch64:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

N, bit [31]

Set to the value of the N condition flag on taking an exception to EL1, and copied to the N condition flag on executing an exception return operation in EL1.

Z, bit [30]

Set to the value of the Z condition flag on taking an exception to EL1, and copied to the Z condition flag on executing an exception return operation in EL1.

C, bit [29]

Set to the value of the C condition flag on taking an exception to EL1, and copied to the C condition flag on executing an exception return operation in EL1.

V, bit [28]

Set to the value of the V condition flag on taking an exception to EL1, and copied to the V condition flag on executing an exception return operation in EL1.

Bits [27:23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

Bits [19:10]

Reserved, RES0.

D, bit [9]

Process state D mask. The possible values of this bit are:

- 0 Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
- 1 Watchpoint, Breakpoint, and Software step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

A, bit [8]

SError (System Error) mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 0 Exception taken from AArch64.

M[3:0], bits [3:0]

AArch64 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	EL0t
0b0100	EL1t
0b0101	EL1h

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
 - 0 means the SP is always SP0.
 - 1 means the exception SP is determined by the EL.

Accessing the SPSR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_EL1	11	000	0100	0000	000
SPSR_EL12	11	101	0100	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_EL1	x	x	0	-	RW	n/a	RW
SPSR_EL1	0	0	1	-	RW	RW	RW
SPSR_EL1	0	1	1	-	n/a	RW	RW
SPSR_EL1	1	0	1	-	RW	SPSR_EL2	RW
SPSR_EL1	1	1	1	-	n/a	SPSR_EL2	RW
SPSR_EL12	x	x	0	-	-	n/a	-
SPSR_EL12	0	0	1	-	-	-	-
SPSR_EL12	0	1	1	-	n/a	-	-
SPSR_EL12	1	0	1	-	-	RW	RW
SPSR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic SPSR_EL1 or SPSR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.42 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL2.

Configurations

AArch64 System register SPSR_EL2 is architecturally mapped to AArch32 System register [SPSR_hyp](#).

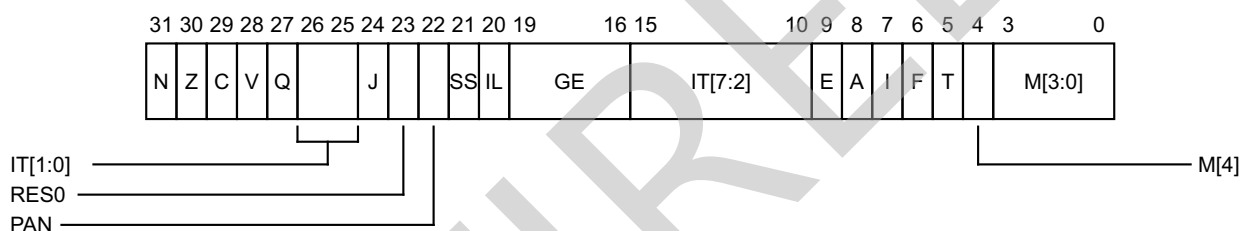
Attributes

SPSR_EL2 is a 32-bit register.

Field descriptions

The SPSR_EL2 bit assignments are:

When exception taken from AArch32:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Hyp mode, and copied to [CPSR.N](#) on executing an exception return operation in Hyp mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Hyp mode, and copied to [CPSR.Z](#) on executing an exception return operation in Hyp mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Hyp mode, and copied to [CPSR.C](#) on executing an exception return operation in Hyp mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Hyp mode, and copied to [CPSR.V](#) on executing an exception return operation in Hyp mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to Hyp mode, and copied to [CPSR.PAN](#) on executing an exception return operation in Hyp mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTL.R.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

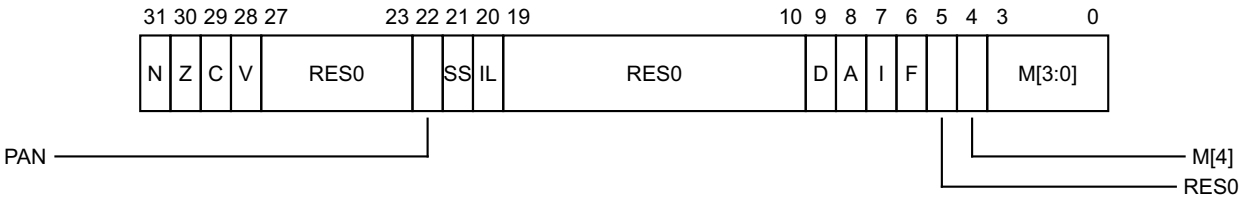
M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries](#) on page B12-558.

When exception taken from AArch64:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

N, bit [31]

Set to the value of the N condition flag on taking an exception to EL2, and copied to the N condition flag on executing an exception return operation in EL2.

Z, bit [30]

Set to the value of the Z condition flag on taking an exception to EL2, and copied to the Z condition flag on executing an exception return operation in EL2.

C, bit [29]

Set to the value of the C condition flag on taking an exception to EL2, and copied to the C condition flag on executing an exception return operation in EL2.

V, bit [28]

Set to the value of the V condition flag on taking an exception to EL2, and copied to the V condition flag on executing an exception return operation in EL2.

Bits [27:23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

Bits [19:10]

Reserved, RES0.

D, bit [9]

Process state D mask. The possible values of this bit are:

- 0 Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
- 1 Watchpoint, Breakpoint, and Software step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

A, bit [8]

SError (System Error) mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 0 Exception taken from AArch64.

M[3:0], bits [3:0]

AArch64 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	EL0t
0b0100	EL1t
0b0101	EL1h
0b1000	EL2t
0b1001	EL2h

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
 - 0 means the SP is always SP0.
 - 1 means the exception SP is determined by the EL.

Accessing the SPSR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_EL2	11	100	0100	0000	000
SPSR_EL1	11	000	0100	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_EL2	x	x	0	-	-	n/a	RW
SPSR_EL2	0	0	1	-	-	RW	RW
SPSR_EL2	0	1	1	-	n/a	RW	RW
SPSR_EL2	1	0	1	-	-	RW	RW
SPSR_EL2	1	1	1	-	n/a	RW	RW
SPSR_EL1	x	x	0	-	SPSR_EL1	n/a	SPSR_EL1
SPSR_EL1	0	0	1	-	SPSR_EL1	SPSR_EL1	SPSR_EL1
SPSR_EL1	0	1	1	-	n/a	SPSR_EL1	SPSR_EL1
SPSR_EL1	1	0	1	-	SPSR_EL1	RW	SPSR_EL1
SPSR_EL1	1	1	1	-	n/a	RW	SPSR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic SPSR_EL2 or SPSR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.43 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL3.

Configurations

AArch64 System register SPSR_EL3 can be mapped to AArch32 System register [SPSR_mon](#), but this is not architecturally mandated.

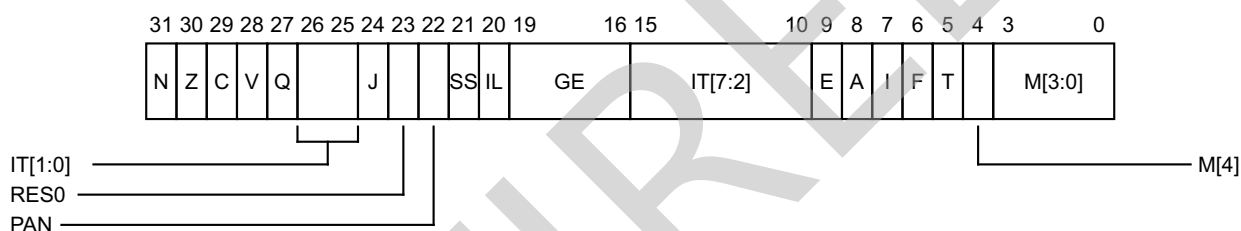
Attributes

SPSR_EL3 is a 32-bit register.

Field descriptions

The SPSR_EL3 bit assignments are:

When exception taken from AArch32:



An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Monitor mode, and copied to [CPSR.N](#) on executing an exception return operation in Monitor mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Monitor mode, and copied to [CPSR.Z](#) on executing an exception return operation in Monitor mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Monitor mode, and copied to [CPSR.C](#) on executing an exception return operation in Monitor mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Monitor mode, and copied to [CPSR.V](#) on executing an exception return operation in Monitor mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to Monitor mode, and copied to [CPSR.PAN](#) on executing an exception return operation in Monitor mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTL.R.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

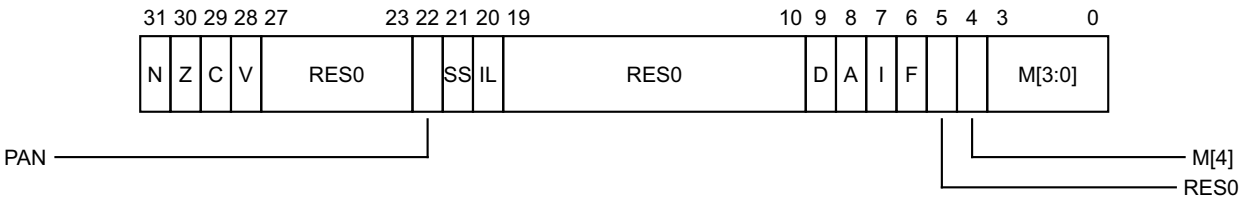
M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

When exception taken from AArch64:



An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

N, bit [31]

Set to the value of the N condition flag on taking an exception to EL3, and copied to the N condition flag on executing an exception return operation in EL3.

Z, bit [30]

Set to the value of the Z condition flag on taking an exception to EL3, and copied to the Z condition flag on executing an exception return operation in EL3.

C, bit [29]

Set to the value of the C condition flag on taking an exception to EL3, and copied to the C condition flag on executing an exception return operation in EL3.

V, bit [28]

Set to the value of the V condition flag on taking an exception to EL3, and copied to the V condition flag on executing an exception return operation in EL3.

Bits [27:23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

Bits [19:10]

Reserved, RES0.

D, bit [9]

Process state D mask. The possible values of this bit are:

- 0 Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
- 1 Watchpoint, Breakpoint, and Software step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

A, bit [8]

SError (System Error) mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 0 Exception taken from AArch64.

M[3:0], bits [3:0]

AArch64 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	EL0t
0b0100	EL1t
0b0101	EL1h
0b1000	EL2t
0b1001	EL2h
0b1100	EL3t
0b1101	EL3h

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
 - 0 means the SP is always SP0.
 - 1 means the exception SP is determined by the EL.

Accessing the SPSR_EL3

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_EL3	11	110	0100	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_EL3	x	x	0	-	-	n/a	RW
SPSR_EL3	x	0	1	-	-	-	RW
SPSR_EL3	x	1	1	-	n/a	-	RW

RETIRED

B12.2.44 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configurations

AArch64 System register SPSR_fiq is architecturally mapped to AArch32 System register [SPSR_fiq](#).

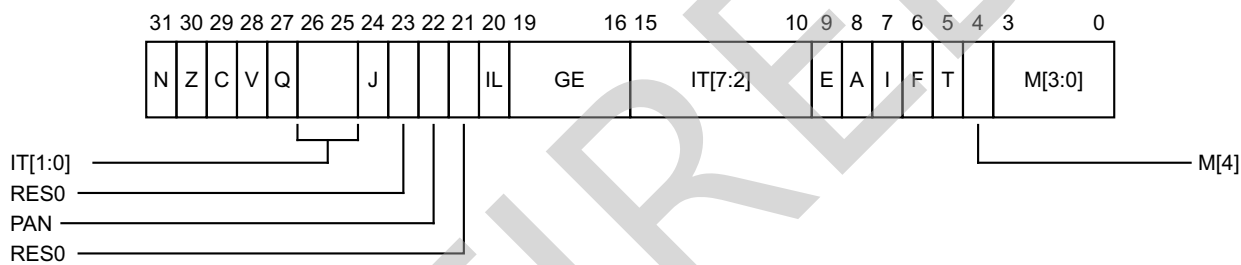
If EL1 does not support execution in AArch32, this register is RES0.

Attributes

SPSR_fiq is a 32-bit register.

Field descriptions

The SPSR_fiq bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to FIQ mode, and copied to [CPSR.N](#) on executing an exception return operation in FIQ mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to FIQ mode, and copied to [CPSR.Z](#) on executing an exception return operation in FIQ mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to FIQ mode, and copied to [CPSR.C](#) on executing an exception return operation in FIQ mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to FIQ mode, and copied to [CPSR.V](#) on executing an exception return operation in FIQ mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to FIQ mode, and copied to [CPSR.PAN](#) on executing an exception return operation in FIQ mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

Accessing the SPSR_fiq

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_fiq	11	100	0100	0011	011

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_fiq	x	x	0	-	-	n/a	RW
SPSR_fiq	x	0	1	-	-	RW	RW
SPSR_fiq	x	1	1	-	n/a	RW	RW

B12.2.45 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configurations

AArch64 System register SPSR_irq is architecturally mapped to AArch32 System register [SPSR_irq](#).

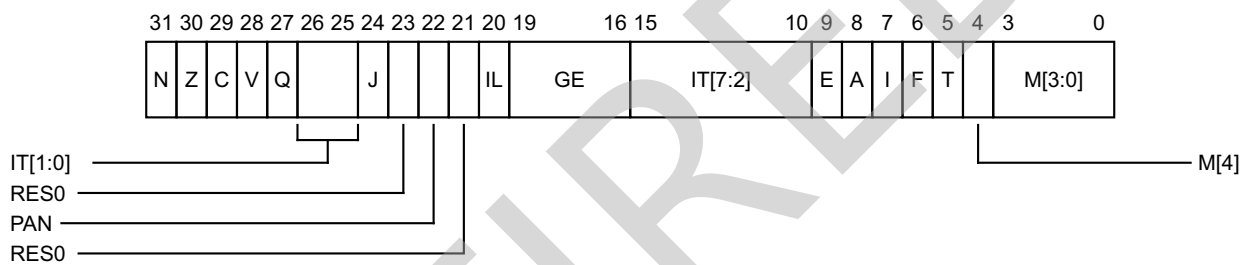
If EL1 does not support execution in AArch32, this register is RES0.

Attributes

SPSR_irq is a 32-bit register.

Field descriptions

The SPSR_irq bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to IRQ mode, and copied to [CPSR.N](#) on executing an exception return operation in IRQ mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to IRQ mode, and copied to [CPSR.Z](#) on executing an exception return operation in IRQ mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to IRQ mode, and copied to [CPSR.C](#) on executing an exception return operation in IRQ mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to IRQ mode, and copied to [CPSR.V](#) on executing an exception return operation in IRQ mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to IRQ mode, and copied to [CPSR.PAN](#) on executing an exception return operation in IRQ mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

Accessing the SPSR_irq

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_irq	11	100	0100	0011	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_irq	x	x	0	-	-	n/a	RW
SPSR_irq	x	0	1	-	-	RW	RW
SPSR_irq	x	1	1	-	n/a	RW	RW

B12.2.46 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configurations

AArch64 System register SPSR_und is architecturally mapped to AArch32 System register [SPSR_und](#).

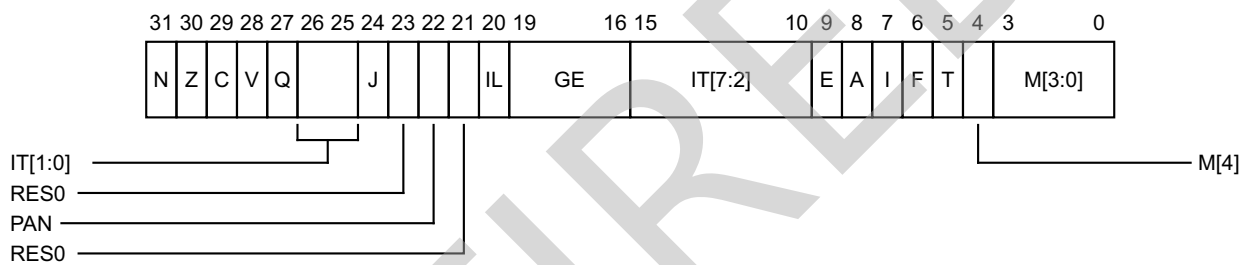
If EL1 does not support execution in AArch32, this register is RES0.

Attributes

SPSR_und is a 32-bit register.

Field descriptions

The SPSR_und bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Undefined mode, and copied to [CPSR.N](#) on executing an exception return operation in Undefined mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Undefined mode, and copied to [CPSR.Z](#) on executing an exception return operation in Undefined mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Undefined mode, and copied to [CPSR.C](#) on executing an exception return operation in Undefined mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Undefined mode, and copied to [CPSR.V](#) on executing an exception return operation in Undefined mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of **CPSR.PAN** on taking an exception to Undefined mode, and copied to **CPSR.PAN** on executing an exception return operation in Undefined mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of **PSTATE.IL** immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- **IT[7:5]** holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- **IT[4:0]** encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the **SCTLR.EE** bit is defined by a configuration input signal, that value also applies to the **CPSR.E** bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

Accessing the SPSR_und

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
SPSR_und	11	100	0100	0011	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_und	x	x	0	-	-	n/a	RW
SPSR_und	x	0	1	-	-	RW	RW
SPSR_und	x	1	1	-	n/a	RW	RW

B12.2.47 TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

Determines which of the Translation Table Base Registers defines the base address for a translation table walk required for the stage 1 translation of a memory access from EL0 or EL1. Also controls the translation table format and holds cacheability and shareability information.

This register is used when [HCR_EL2.E2H](#) is 0.

Note

When [HCR_EL2.E2H](#) is 1, [TCR_EL2](#) is used.

Configurations

AArch64 System register TCR_EL1[31:0] is architecturally mapped to AArch32 System register TTBCR.

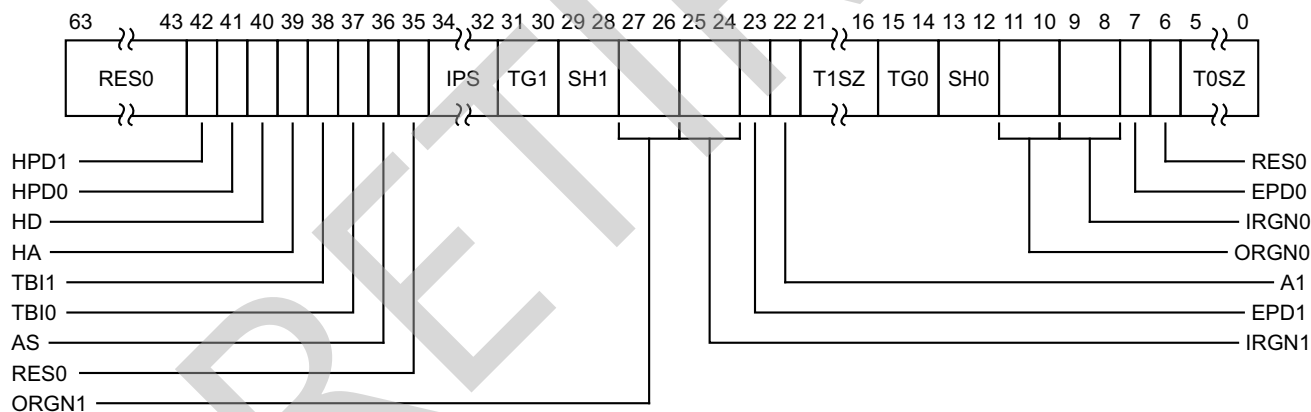
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

TCR_EL1 is a 64-bit register.

Field descriptions

The TCR_EL1 bit assignments are:



Any of the bits in TCR_EL1 are permitted to be cached in a TLB.

Bits [63:43]

Reserved, RES0.

HPD1, bit [42] (In ARMv8.1)

Hierarchical Permission Disables. This affects the hierarchical control bits, APTTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by [TTBR1_EL1](#).

0 Hierarchical Permissions are enabled.

1 Hierarchical Permissions are disabled.

When disabled, the behavior is as if the bits are zero.

Bit [42] (In ARMv8.0)

Reserved, RES0.

HPD0, bit [41] (In ARMv8.1)

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by [TTBR0_EL1](#).

0 Hierarchical Permissions are enabled.

1 Hierarchical Permissions are disabled.

When disabled, the behavior is as if the bits are zero.

Bit [41] (In ARMv8.0)

Reserved, RES0.

HD, bit [40] (In ARMv8.1)

Hardware management of dirty state in stage 1 translations from EL0 and EL1.

0 Stage 1 hardware management of dirty state disabled.

1 Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

Bit [40] (In ARMv8.0)

Reserved, RES0.

HA, bit [39] (In ARMv8.1)

Hardware Access flag update in stage 1 translations from EL0 and EL1.

0 Stage 1 Access flag update disabled.

1 Stage 1 Access flag update enabled.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

Bit [39] (In ARMv8.0)

Reserved, RES0.

TBI1, bit [38]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the [TTBR1_EL1](#) region, or ignored and used for tagged addresses. Defined values are:

0 Top Byte used in the address calculation.

1 Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by [TTBR1_EL1](#). It has an effect whether the EL1&0 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI1 is 1 and bit [55] of the target address is 1, caused by:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

In these cases bits [63:56] of the address are also set to 1 before it is stored in the PC.

TBI0, bit [37]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the [TTBR0_EL1](#) region, or ignored and used for tagged addresses. Defined values are:

0 Top Byte used in the address calculation.

1 Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by [TTBR0_EL1](#). It has an effect whether the EL1&0 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI0 is 1 and bit [55] of the target address is 0, caused by:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

In these cases bits [63:56] of the address are also set to 0 before it is stored in the PC.

AS, bit [36]

ASID Size. Defined values are:

- | | |
|---|--|
| 0 | 8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB. |
| 1 | 16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB. |

If the implementation has only 8 bits of ASID, this field is RES0.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

Intermediate Physical Address Size.

- | | |
|-----|-----------------|
| 000 | 32 bits, 4GB. |
| 001 | 36 bits, 64GB. |
| 010 | 40 bits, 1TB. |
| 011 | 42 bits, 4TB. |
| 100 | 44 bits, 16TB. |
| 101 | 48 bits, 256TB. |

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

TG1, bits [31:30]

Granule size for the [TTBR1_EL1](#).

- | | |
|----|------|
| 01 | 16KB |
| 10 | 4KB |
| 11 | 64KB |

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using [TTBR1_EL1](#).

Defined values are:

- | | |
|----|-----------------|
| 00 | Non-shareable |
| 10 | Outer Shareable |

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page B12-558.

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using [TTBR1_EL1](#).

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using [TTBR1_EL1](#).

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

EPD1, bit [23]

Translation table walk disable for translations using [TTBR1_EL1](#). This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using [TTBR1_EL1](#). The encoding of this bit is:

0	Perform translation table walks using TTBR1_EL1 .
1	A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.

A1, bit [22]

Selects whether [TTBR0_EL1](#) or [TTBR1_EL1](#) defines the ASID. The encoding of this bit is:

0	TTBR0_EL1 .ASID defines the ASID.
1	TTBR1_EL1 .ASID defines the ASID.

T1SZ, bits [21:16]

The size offset of the memory region addressed by [TTBR1_EL1](#). The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

TG0, bits [15:14]

Granule size for the [TTBR0_EL1](#).

00	4KB
01	64KB
10	16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using [TTBR0_EL1](#).

00	Non-shareable
10	Outer Shareable
11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page B12-558.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using [TTBR0_EL1](#).

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using [TTBR0_EL1](#).

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

EPD0, bit [7]

Translation table walk disable for translations using [TTBR0_EL1](#). This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using [TTBR0_EL1](#). The encoding of this bit is:

0	Perform translation table walks using TTBR0_EL1 .
1	A TLB miss on an address that is translated using TTBR0_EL1 generates a Translation fault. No translation table walk is performed.

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by [TTBR0_EL1](#). The region size is $2^{(64-T0SZ)}$ bytes. The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Accessing the TCR_EL1

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```


This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TCR_EL1	11	000	0010	0000	010
TCR_EL12	11	101	0010	0000	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TCR_EL1	x	x	0	-	RW	n/a	RW
TCR_EL1	0	0	1	-	RW	RW	RW
TCR_EL1	0	1	1	-	n/a	RW	RW
TCR_EL1	1	0	1	-	RW	TCR_EL2	RW
TCR_EL1	1	1	1	-	n/a	TCR_EL2	RW
TCR_EL12	x	x	0	-	-	n/a	-
TCR_EL12	0	0	1	-	-	-	-
TCR_EL12	0	1	1	-	n/a	-	-
TCR_EL12	1	0	1	-	-	RW	RW
TCR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic TCR_EL1 or TCR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.48 TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

Purpose

When [HCR_EL2.E2H](#) is 0, controls translation table walks required for the stage 1 translation of memory accesses from EL2.

When [HCR_EL2.E2H](#) is 1, determines which of the Translation Table Base Registers defines the base address for a translation table walk required for the stage 1 translation of a memory access from EL0 or EL2.

It also controls the translation table format, and holds cacheability and shareability information.

Configurations

AArch64 System register TCR_EL2 is architecturally mapped to AArch32 System register HTCR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

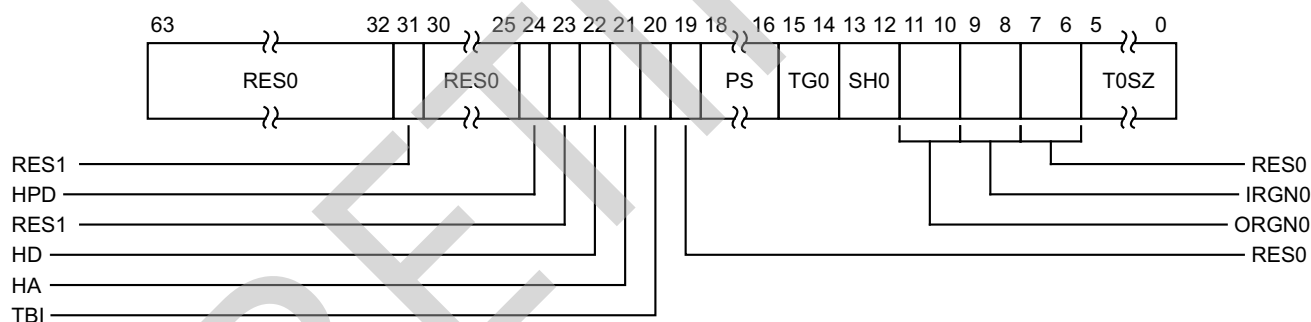
Attributes

TCR_EL2 is a 64-bit register.

Field descriptions

The TCR_EL2 bit assignments are:

When [HCR_EL2.E2H](#)==0:



Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

This format applies in all ARMv8.0 implementations.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

Bits [30:25]

Reserved, RES0.

HPD, bit [24]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by [TTBR0_EL2](#).

0 Hierarchical Permissions are enabled.

1 Hierarchical Permissions are disabled.

Note

In this case bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE, and are no longer reserved, allowing them to be used by software.

When disabled, the behavior is as if the bits are zero.

Bit [23]

Reserved, RES1.

HD, bit [22]

Hardware management of dirty state in stage 1 translations from EL2.

0 Stage 1 hardware management of dirty state disabled.

1 Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

HA, bit [21]

Hardware Access flag update in stage 1 translations from EL2.

0 Stage 1 Access flag update disabled.

1 Stage 1 Access flag update enabled.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

TBI, bit [20]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the [TTBR0_EL2](#) region, or ignored and used for tagged addresses.

0 Top Byte used in the address calculation.

1 Top Byte ignored in the address calculation.

This affects addresses generated in EL2 using AArch64 where the address would be translated by tables pointed to by [TTBR0_EL2](#). It has an effect whether the EL2 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI is 1, caused by:

- A branch or procedure return within EL2.
- An exception taken to EL2.
- An exception return to EL2.

In these cases bits [63:56] of the address are set to 0 before it is stored in the PC.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

000 32 bits, 4GB.

001 36 bits, 64GB.

010 40 bits, 1TB.

011 42 bits, 4TB.

100 44 bits, 16TB.

101 48 bits, 256TB.

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

TG0, bits [15:14]

Granule size for the [TTBR0_EL2](#).

00	4KB
01	64KB
10	16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using [TTBR0_EL2](#).

00	Non-shareable
10	Outer Shareable
11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page B12-558.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using [TTBR0_EL2](#).

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using [TTBR0_EL2](#).

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

Bits [7:6]

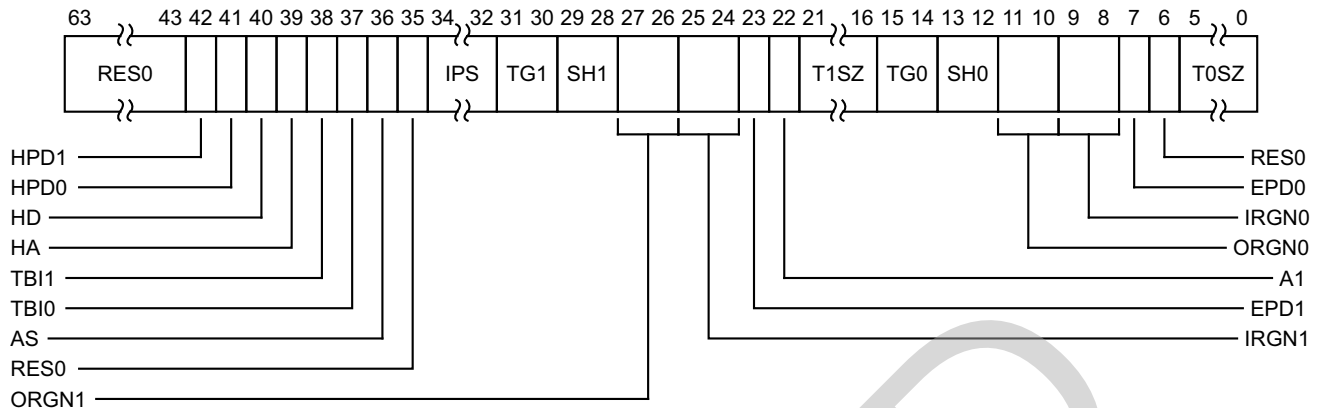
Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by [TTBR0_EL2](#). The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

When HCR_EL2.E2H==1:



Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

Bits [63:43]

Reserved, RES0.

HPD1, bit [42]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by [TTBR1_EL2](#).

0 Hierarchical Permissions are enabled.

1 Hierarchical Permissions are disabled.

When disabled, the behavior is as if the bits are zero.

HPD0, bit [41]

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by [TTBR0_EL2](#).

0 Hierarchical Permissions are enabled.

1 Hierarchical Permissions are disabled.

When disabled, the behavior is as if the bits are zero.

HD, bit [40]

Hardware management of dirty state in stage 1 translations from EL2.

0 Stage 1 hardware management of dirty state disabled.

1 Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

HA, bit [39]

Hardware Access flag update in stage 1 translations from EL2.

0 Stage 1 Access flag update disabled.

1 Stage 1 Access flag update enabled.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

TBI1, bit [38]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the [TTBR1_EL2](#) region, or ignored and used for tagged addresses. Defined values are:

0 Top Byte used in the address calculation.

1 Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by [TTBR1_EL2](#). It has an effect whether the EL2&0 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI1 is 1 and bit [55] of the target address is 1, caused by:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

In these cases bits [63:56] of the address are also set to 1 before it is stored in the PC.

TBI0, bit [37]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the [TTBR0_EL2](#) region, or ignored and used for tagged addresses. Defined values are:

0 Top Byte used in the address calculation.

1 Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by [TTBR0_EL2](#). It has an effect whether the EL2&0 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI0 is 1 and bit [55] of the target address is 0, caused by:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

In these cases bits [63:56] of the address are also set to 0 before it is stored in the PC.

AS, bit [36]

ASID Size. Defined values are:

0 8 bit - the upper 8 bits of [TTBR0_EL2](#) and [TTBR1_EL2](#) are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.

1 16 bit - the upper 16 bits of [TTBR0_EL2](#) and [TTBR1_EL2](#) are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

Intermediate Physical Address Size.

000 32 bits, 4GB.

001 36 bits, 64GB.

010 40 bits, 1TB.

011 42 bits, 4TB.

100 44 bits, 16TB.

101 48 bits, 256TB.

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

TG1, bits [31:30]

Granule size for the [TTBR1_EL2](#).

01	16KB
10	4KB
11	64KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH1, bits [29:28]

Shareability attribute for memory associated with translation table walks using [TTBR1_EL2](#).

Defined values are:

00	Non-shareable
10	Outer Shareable
11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

ORGN1, bits [27:26]

Outer cacheability attribute for memory associated with translation table walks using [TTBR1_EL2](#).

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN1, bits [25:24]

Inner cacheability attribute for memory associated with translation table walks using [TTBR1_EL2](#).

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

EPD1, bit [23]

Translation table walk disable for translations using [TTBR1_EL2](#). This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using [TTBR1_EL2](#). The encoding of this bit is:

0	Perform translation table walks using TTBR1_EL2 .
1	A TLB miss on an address that is translated using TTBR1_EL2 generates a Translation fault. No translation table walk is performed.

A1, bit [22]

Selects whether [TTBR0_EL2](#) or [TTBR1_EL2](#) defines the ASID. The encoding of this bit is:

0	TTBR0_EL2 .ASID defines the ASID.
1	TTBR1_EL2 .ASID defines the ASID.

T1SZ, bits [21:16]

The size offset of the memory region addressed by [TTBR1_EL2](#). The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

TG0, bits [15:14]

Granule size for the [TTBR0_EL2](#).

00	4KB
01	64KB
10	16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using [TTBR0_EL2](#).

00	Non-shareable
10	Outer Shareable
11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page B12-558.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using [TTBR0_EL2](#).

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using [TTBR0_EL2](#).

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

EPD0, bit [7]

Translation table walk disable for translations using [TTBR0_EL2](#). This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using [TTBR0_EL2](#). The encoding of this bit is:

0	Perform translation table walks using TTBR0_EL2 .
1	A TLB miss on an address that is translated using TTBR0_EL2 generates a Translation fault. No translation table walk is performed.

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by [TTBR0_EL2](#). The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Accessing the TCR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TCR_EL2	11	100	0010	0000	010
TCR_EL1	11	000	0010	0000	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TCR_EL2	x	x	0	-	-	n/a	RW
TCR_EL2	0	0	1	-	-	RW	RW
TCR_EL2	0	1	1	-	n/a	RW	RW
TCR_EL2	1	0	1	-	-	RW	RW
TCR_EL2	1	1	1	-	n/a	RW	RW
TCR_EL1	x	x	0	-	TCR_EL1	n/a	TCR_EL1
TCR_EL1	0	0	1	-	TCR_EL1	TCR_EL1	TCR_EL1
TCR_EL1	0	1	1	-	n/a	TCR_EL1	TCR_EL1
TCR_EL1	1	0	1	-	TCR_EL1	RW	TCR_EL1
TCR_EL1	1	1	1	-	n/a	RW	TCR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic TCR_EL2 or TCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.49 TCR_EL3, Translation Control Register (EL3)

The TCR_EL3 characteristics are:

Purpose

Controls translation table walks required for the stage 1 translation of memory accesses from EL3, and holds cacheability and shareability information for the accesses.

Configurations

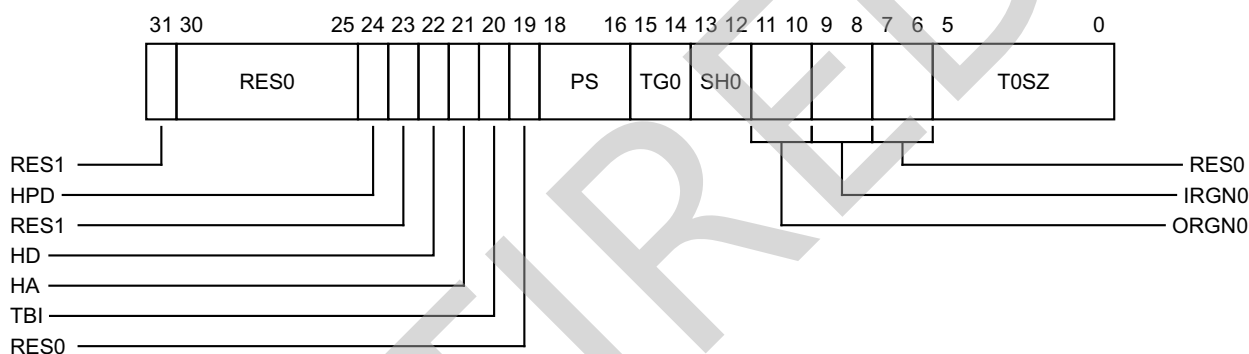
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

TCR_EL3 is a 32-bit register.

Field descriptions

The TCR_EL3 bit assignments are:



Any of the bits in TCR_EL3 are permitted to be cached in a TLB.

Bit [31]

Reserved, RES1.

Bits [30:25]

Reserved, RES0.

HPD, bit [24] (In ARMv8.1)

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL3.

0 Hierarchical Permissions are enabled.

1 Hierarchical Permissions are disabled.

Note

In this case bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE, and are no longer reserved, allowing them to be used by software.

When disabled, the behavior is as if the bits are zero.

Bit [24] (In ARMv8.0)

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22] (In ARMv8.1)

Hardware management of dirty state in stage 1 translations from EL3.

0 Stage 1 hardware management of dirty state disabled.

1 Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

Bit [22] (In ARMv8.0)

Reserved, RES0.

HA, bit [21] (In ARMv8.1)

Hardware Access flag update in stage 1 translations from EL3.

0 Stage 1 Access flag update disabled.

1 Stage 1 Access flag update enabled.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

Bit [21] (In ARMv8.0)

Reserved, RES0.

TBI, bit [20]

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR0_EL3 region, or ignored and used for tagged addresses.

0 Top Byte used in the address calculation.

1 Top Byte ignored in the address calculation.

This affects addresses generated in EL3 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL3. It has an effect whether the EL3 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI is 1, caused by:

- A branch or procedure return within EL3.
- A exception taken to EL3.
- An exception return to EL3.

In these cases bits [63:56] of the address are set to 0 before it is stored in the PC.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

000 32 bits, 4GB.

001 36 bits, 64GB.

010 40 bits, 1TB.

011 42 bits, 4TB.

100 44 bits, 16TB.

101 48 bits, 256TB.

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

TG0, bits [15:14]

Granule size for the TTBR0_EL3.

00	4KB
01	64KB
10	16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using TTBR0_EL3.

00	Non-shareable
10	Outer Shareable
11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

Bits [7:6]

Reserved, RES0.

T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL3. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Accessing the TCR_EL3

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TCR_EL3	11	110	0010	0000	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TCR_EL3	x	x	0	-	-	n/a	RW
TCR_EL3	0	0	1	-	-	-	RW
TCR_EL3	0	1	1	-	n/a	-	RW
TCR_EL3	1	0	1	-	-	-	RW
TCR_EL3	1	1	1	-	n/a	-	RW

B12.2.50 TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

Purpose

Holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL0 and EL1.

This register is used when [HCR_EL2.E2H](#) is 0.

Note

When [HCR_EL2.E2H](#) is 1, [TTBR0_EL2](#) is used.

Configurations

AArch64 System register TTBR0_EL1 is architecturally mapped to AArch32 System register TTBR0.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

TTBR0_EL1 is a 64-bit register.

Field descriptions

The TTBR0_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

ASID, bits [63:48]

An ASID for the translation table base address. The [TCR_EL1.A1](#) field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

BADDR, bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of [TCR_EL1.T0SZ](#), the stage of translation, and the translation granule size.

Accessing the TTBR0_EL1

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TTBR0_EL1	11	000	0010	0000	000
TTBR0_EL12	11	101	0010	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TTBR0_EL1	x	x	0	-	RW	n/a	RW
TTBR0_EL1	0	0	1	-	RW	RW	RW
TTBR0_EL1	0	1	1	-	n/a	RW	RW
TTBR0_EL1	1	0	1	-	RW	TTBR0_EL2	RW
TTBR0_EL1	1	1	1	-	n/a	TTBR0_EL2	RW
TTBR0_EL12	x	x	0	-	-	n/a	-
TTBR0_EL12	0	0	1	-	-	-	-
TTBR0_EL12	0	1	1	-	n/a	-	-
TTBR0_EL12	1	0	1	-	-	RW	RW
TTBR0_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic TTBR0_EL1 or TTBR0_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.51 TTBR0_EL2, Translation Table Base Register 0 (EL2)

The TTBR0_EL2 characteristics are:

Purpose

When [HCR_EL2.E2H](#) is 0, holds the base address of the translation table for the stage 1 translation of memory accesses from EL2.

When [HCR_EL2.E2H](#) is 1, holds the base address of translation table 0 for stage 1 of the EL2&0 translation regime.

Configurations

AArch64 System register TTBR0_EL2 is architecturally mapped to AArch32 System register HTTBR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

TTBR0_EL2 is a 64-bit register.

Field descriptions

The TTBR0_EL2 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

ASID, bits [63:48] (In ARMv8.1)

When [HCR_EL2.E2H](#) is 0, this field is RES0.

When [HCR_EL2.E2H](#) is 1, it holds an ASID for the translation table base address. The [TCR_EL2.A1](#) field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

Bits [63:48] (In ARMv8.0)

Reserved, RES0.

BADDR, bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of [TCR_EL2.T0SZ](#), the stage of translation, and the translation granule size.

Accessing the TTBR0_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TTBR0_EL2	11	100	0010	0000	000
TTBR0_EL1	11	000	0010	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TTBR0_EL2	x	x	0	-	-	n/a	RW
TTBR0_EL2	0	0	1	-	-	RW	RW
TTBR0_EL2	0	1	1	-	n/a	RW	RW
TTBR0_EL2	1	0	1	-	-	RW	RW
TTBR0_EL2	1	1	1	-	n/a	RW	RW
TTBR0_EL1	x	x	0	-	TTBR0_EL1	n/a	TTBR0_EL1
TTBR0_EL1	0	0	1	-	TTBR0_EL1	TTBR0_EL1	TTBR0_EL1
TTBR0_EL1	0	1	1	-	n/a	TTBR0_EL1	TTBR0_EL1
TTBR0_EL1	1	0	1	-	TTBR0_EL1	RW	TTBR0_EL1
TTBR0_EL1	1	1	1	-	n/a	RW	TTBR0_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic TTBR0_EL2 or TTBR0_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.52 TTBR1_EL1, Translation Table Base Register 1 (EL1)

The TTBR1_EL1 characteristics are:

Purpose

Holds the base address of translation table 1, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL0 and EL1.

This register is used when [HCR_EL2.E2H](#) is 0.

Note

When [HCR_EL2.E2H](#) is 1, [TTBR1_EL2](#) is used.

Configurations

AArch64 System register TTBR1_EL1 is architecturally mapped to AArch32 System register TTBR1.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

TTBR1_EL1 is a 64-bit register.

Field descriptions

The TTBR1_EL1 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

ASID, bits [63:48]

An ASID for the translation table base address. The [TCR_EL1.A1](#) field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

BADDR, bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of [TCR_EL1.T1SZ](#), the stage of translation, and the translation granule size.

Accessing the TTBR1_EL1

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TTBR1_EL1	11	000	0010	0000	001
TTBR1_EL12	11	101	0010	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TTBR1_EL1	x	x	0	-	RW	n/a	RW
TTBR1_EL1	0	0	1	-	RW	RW	RW
TTBR1_EL1	0	1	1	-	n/a	RW	RW
TTBR1_EL1	1	0	1	-	RW	TTBR1_EL2	RW
TTBR1_EL1	1	1	1	-	n/a	TTBR1_EL2	RW
TTBR1_EL12	x	x	0	-	-	n/a	-
TTBR1_EL12	0	0	1	-	-	-	-
TTBR1_EL12	0	1	1	-	n/a	-	-
TTBR1_EL12	1	0	1	-	-	RW	RW
TTBR1_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic TTBR1_EL1 or TTBR1_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.

B12.2.53 TTBR1_EL2, Translation Table Base Register 1 (EL2)

The TTBR1_EL2 characteristics are:

Purpose

When [HCR_EL2.E2H](#) is 1, holds the base address of translation table 1 for stage 1 of the EL2&0 translation regime.

Note

When [HCR_EL2.E2H](#) is 0, the contents of this register are ignored by the PE, except for a direct read or write of the register.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

TTBR1_EL2 is a 64-bit register.

Field descriptions

The TTBR1_EL2 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

ASID, bits [63:48]

An ASID for the translation table base address. The [TCR_EL2.A1](#) field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

BADDR, bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of [TCR_EL2.T1SZ](#), the stage of translation, and the translation granule size.

Accessing the TTBR1_EL2

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
TTBR1_EL2	11	100	0010	0000	001
TTBR1_EL1	11	000	0010	0000	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
TTBR1_EL2	x	x	0	-	-	n/a	RW
TTBR1_EL2	0	0	1	-	-	RW	RW
TTBR1_EL2	0	1	1	-	n/a	RW	RW
TTBR1_EL2	1	0	1	-	-	RW	RW
TTBR1_EL2	1	1	1	-	n/a	RW	RW
TTBR1_EL1	x	x	0	-	TTBR1_EL1	n/a	TTBR1_EL1
TTBR1_EL1	0	0	1	-	TTBR1_EL1	TTBR1_EL1	TTBR1_EL1
TTBR1_EL1	0	1	1	-	n/a	TTBR1_EL1	TTBR1_EL1
TTBR1_EL1	1	0	1	-	TTBR1_EL1	RW	TTBR1_EL1
TTBR1_EL1	1	1	1	-	n/a	RW	TTBR1_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic TTBR1_EL2 or TTBR1_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.54 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL1.

Configurations

AArch64 System register VBAR_EL1[31:0] is architecturally mapped to AArch32 System register VBAR.

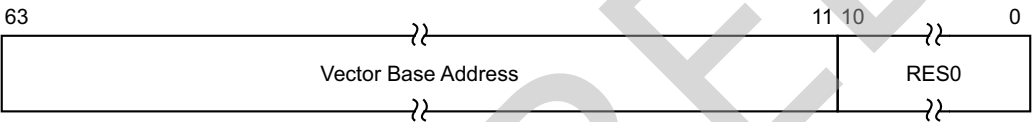
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

VBAR_EL1 is a 64-bit register.

Field descriptions

The VBAR_EL1 bit assignments are:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
VBAR_EL1	11	000	1100	0000	000
VBAR_EL12	11	101	1100	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VBAR_EL1	x	x	0	-	RW	n/a	RW
VBAR_EL1	0	0	1	-	RW	RW	RW
VBAR_EL1	0	1	1	-	n/a	RW	RW
VBAR_EL1	1	0	1	-	RW	VBAR_EL2	RW
VBAR_EL1	1	1	1	-	n/a	VBAR_EL2	RW
VBAR_EL12	x	x	0	-	-	n/a	-
VBAR_EL12	0	0	1	-	-	-	-
VBAR_EL12	0	1	1	-	n/a	-	-
VBAR_EL12	1	0	1	-	-	RW	RW
VBAR_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic VBAR_EL1 or VBAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.55 VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL2.

Configurations

AArch64 System register VBAR_EL2[31:0] is architecturally mapped to AArch32 System register HVBAR.

If EL2 is not implemented, this register is RES0 from EL3.

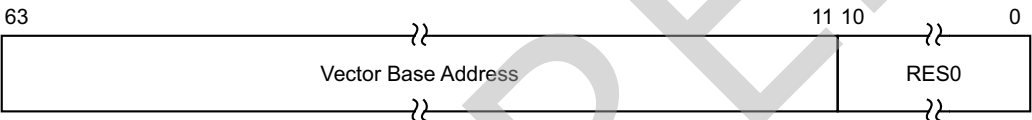
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

VBAR_EL2 is a 64-bit register.

Field descriptions

The VBAR_EL2 bit assignments are:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
VBAR_EL2	11	100	1100	0000	000
VBAR_EL1	11	000	1100	0000	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VBAR_EL2	x	x	0	-	-	n/a	RW
VBAR_EL2	0	0	1	-	-	RW	RW
VBAR_EL2	0	1	1	-	n/a	RW	RW
VBAR_EL2	1	0	1	-	-	RW	RW
VBAR_EL2	1	1	1	-	n/a	RW	RW
VBAR_EL1	x	x	0	-	VBAR_EL1	n/a	VBAR_EL1
VBAR_EL1	0	0	1	-	VBAR_EL1	VBAR_EL1	VBAR_EL1
VBAR_EL1	0	1	1	-	n/a	VBAR_EL1	VBAR_EL1
VBAR_EL1	1	0	1	-	VBAR_EL1	RW	VBAR_EL1
VBAR_EL1	1	1	1	-	n/a	RW	VBAR_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic VBAR_EL2 or VBAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.2.56 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

Purpose

Controls the translation table walks required for the stage 2 translation of memory accesses from Non-secure EL0 and EL1, and holds cacheability and shareability information for the accesses.

Configurations

AArch64 System register VTCR_EL2 is architecturally mapped to AArch32 System register VTCR.

If EL2 is not implemented, this register is RES0 from EL3.

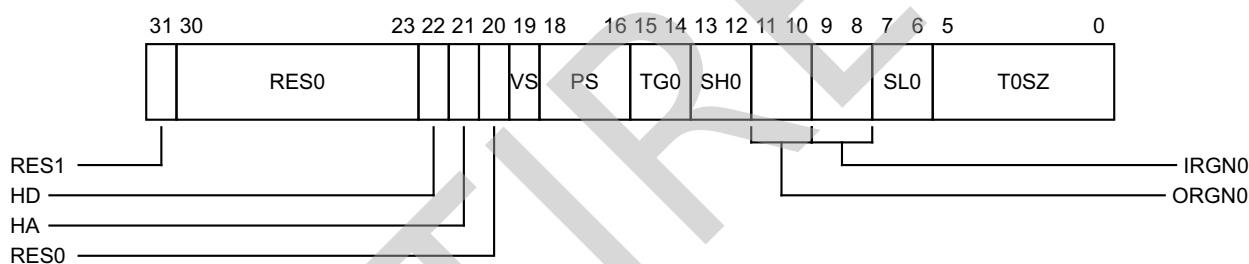
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

VTCR_EL2 is a 32-bit register.

Field descriptions

The VTCR_EL2 bit assignments are:



Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Bit [31]

Reserved, RES1.

Bits [30:23]

Reserved, RES0.

HD, bit [22] (In ARMv8.1)

Hardware management of dirty state in stage 2 translations from Non-secure EL0 and EL1.

0 Stage 2 hardware management of dirty state disabled.

1 Stage 2 hardware management of dirty state enabled, only if the HA bit is also set to 1.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

Bit [22] (In ARMv8.0)

Reserved, RES0.

HA, bit [21] (In ARMv8.1)

Hardware Access flag update in stage 2 translations from Non-secure EL0 and EL1.

0 Stage 2 Access flag update disabled.

1 Stage 2 Access flag update enabled.

Implementation of this bit is OPTIONAL, and, if not implemented, this bit is RES0.

Bit [21] (In ARMv8.0)

Reserved, RES0.

Bit [20]

Reserved, RES0.

VS, bit [19] (In ARMv8.1)

In ARMv8.1, VMID Size

- | | |
|---|---|
| 0 | 8 bit - the upper 8 bits of VTTBR_EL2 are ignored by the hardware, and treated as if they are all zeros, for every purpose except when reading back the register. |
| 1 | 16 bit - the upper 8 bits of VTTBR_EL2 are used for allocation and matching in the TLB. |

If the implementation only supports an 8-bit VMID, this field is RES0.

In ARMv8.0, this bit is RES0.

Bit [19] (In ARMv8.0)

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

- | | |
|-----|-----------------|
| 000 | 32 bits, 4GB. |
| 001 | 36 bits, 64GB. |
| 010 | 40 bits, 1TB. |
| 011 | 42 bits, 4TB. |
| 100 | 44 bits, 16TB. |
| 101 | 48 bits, 256TB. |

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

TG0, bits [15:14]

Granule size for the [VTTBR_EL2](#).

- | | |
|----|------|
| 00 | 4KB |
| 01 | 64KB |
| 10 | 16KB |

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using [VTTBR_EL2](#).

- | | |
|----|-----------------|
| 00 | Non-shareable |
| 10 | Outer Shareable |
| 11 | Inner Shareable |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page B12-558.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using [VTTBR_EL2](#).

00	Normal memory, Outer Non-cacheable
01	Normal memory, Outer Write-Back Write-Allocate Cacheable
10	Normal memory, Outer Write-Through Cacheable
11	Normal memory, Outer Write-Back no Write-Allocate Cacheable

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using [VTTBR_EL2](#).

00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

SL0, bits [7:6]

Starting level of the [VTCR_EL2](#) addressed region. The meaning of this field depends on the value of [VTCR_EL2.TG0](#) (the granule size).

00	If TG0 is 00 (4KB granule), start at level 2. If TG0 is 10 (16KB granule) or 01 (64KB granule), start at level 3.
01	If TG0 is 00 (4KB granule), start at level 1. If TG0 is 10 (16KB granule) or 01 (64KB granule), start at level 2.
10	If TG0 is 00 (4KB granule), start at level 0. If TG0 is 10 (16KB granule) or 01 (64KB granule), start at level 1.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of T0SZ, then a stage 2 level 0 Translation fault is generated.

T0SZ, bits [5:0]

The size offset of the memory region addressed by [VTTBR_EL2](#). The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

If this field is programmed to a value that is not consistent with the programming of SL0 then a stage 2 level 0 Translation fault is generated.

Accessing the VTCR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
VTCR_EL2	11	100	0010	0001	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VTCR_EL2	x	x	0	-	-	n/a	RW
VTCR_EL2	x	0	1	-	-	RW	RW
VTCR_EL2	x	1	1	-	n/a	RW	RW

RETIRED

B12.2.57 VTTBR_EL2, Virtualization Translation Table Base Register

The VTTBR_EL2 characteristics are:

Purpose

Holds the base address of the translation table for the stage 2 translation of memory accesses from Non-secure EL0 and EL1.

Configurations

AArch64 System register VTTBR_EL2 is architecturally mapped to AArch32 System register VTTBR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

VTTBR_EL2 is a 64-bit register.

Field descriptions

The VTTBR_EL2 bit assignments are:



Any of the fields in this register are permitted to be cached in a TLB.

VMID, bits [63:48] (In ARMv8.1)

The VMID for the translation table.

It is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

If the implementation has an 8 bit VMID, then the upper 8 bits of this field are RES0.

If the implementation has a 16 bit VMID, then:

- The [VTCR_EL2.VS](#) bit selects whether the upper 8 bits of this field are ignored by the hardware for every purpose except reading back the register, or whether they are used for allocation and matching in the TLB.
- The 16 bit VMID is only supported when EL2 is using AArch64. This means the hardware must ignore these bits when EL2 is using AArch32.

Bits [63:56] (In ARMv8.0)

Reserved, RES0.

VMID, bits [55:48] (In ARMv8.0)

The VMID for the translation table.

BADDR, bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of [VTCR_EL2.T0SZ](#), the stage of translation, and the translation granule size.

Accessing the VTTBR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
VTTBR_EL2	11	100	0010	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VTTBR_EL2	x	x	0	-	-	n/a	RW
VTTBR_EL2	x	0	1	-	-	RW	RW
VTTBR_EL2	x	1	1	-	n/a	RW	RW

B12.3 Debug registers

This section lists the ARMv8.1 Debug System registers in AArch64 state, in alphabetic order.

RETIRED

B12.3.1 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register [DBGBVR<n>_EL1](#).

Configurations

AArch64 System register DBGBCR<n>_EL1 is architecturally mapped to AArch32 System register [DBGBCR<n>](#).

AArch64 System register DBGBCR<n>_EL1 is architecturally mapped to External register [DBGBCR<n>_EL1](#).

If breakpoint n is not implemented then this register is unallocated.

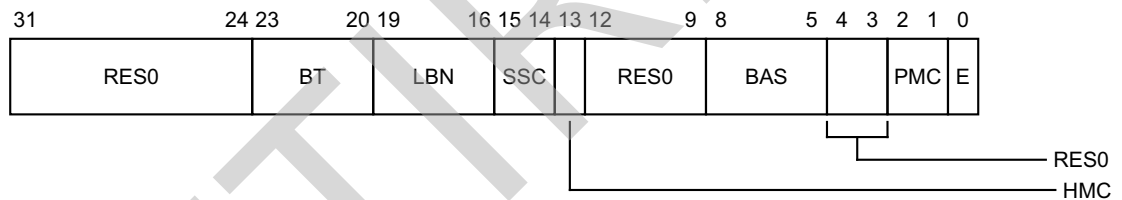
This register is in the Cold reset domain. On a Cold reset RW fields in this register reset to architecturally UNKNOWN values. The register is not affected by a Warm reset.

Attributes

DBGBCR<n>_EL1 is a 32-bit register.

Field descriptions

The DBGBCR<n>_EL1 bit assignments are:



Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

- 0000 Unlinked address match.
- 0001 Linked address match.
- 0010 Unlinked Context ID match.
- 0011 Linked Context ID match.
- 0110 Unlinked [CONTEXTIDR_EL1](#) match (ARMv8.1).
- 0111 Linked [CONTEXTIDR_EL1](#) match (ARMv8.1).
- 1000 Unlinked VMID match.
- 1001 Linked VMID match.
- 1010 Unlinked VMID and Context ID match.
- 1011 Linked VMID and Context ID match.
- 1100 Unlinked [CONTEXTIDR_EL2](#) match (ARMv8.1).
- 1101 Linked [CONTEXTIDR_EL2](#) match (ARMv8.1).
- 1110 Unlinked Full Context ID match (ARMv8.1).
- 1111 Linked Full Context ID match (ARMv8.1).

The field breaks down as follows:

- BT[3:1]: Base type.

000	Match address. DBGBVR<n>_EL1 is the address of an instruction.
001	Match Context ID. DBGBVR<n>_EL1 .ContextID is a Context ID compared against CONTEXTIDR_EL1 in ARMv8.0, and in ARMv8.1 when not in a Host OS or a Host Application. In ARMv8.1, when in a Host OS or Host Application, the Context ID is compared against CONTEXTIDR_EL1 .
011	Match CONTEXTIDR_EL1 . DBGBVR<n>_EL1 .ContextID is a Context ID compared against CONTEXTIDR_EL1 .
100	Match VMID. DBGBVR<n>_EL1 .VMID is a VMID compared against VTTBR_EL2 .VMID.
101	Match VMID and Context ID. DBGBVR<n>_EL1 .ContextID is a Context ID compared against CONTEXTIDR_EL1 , and DBGBVR<n>_EL1 .VMID is a VMID compared against VTTBR_EL2 .VMID.
110	Match CONTEXTIDR_EL2 . DBGBVR<n>_EL1 .ContextID2 is a Context ID compared against CONTEXTIDR_EL2 .
111	Match Full Context ID. DBGBVR<n>_EL1 .ContextID is compared against CONTEXTIDR_EL1 , and DBGBVR<n>_EL1 .ContextID2 is compared against CONTEXTIDR_EL2 .
- BT[0]: Enable linking.

All other values are reserved. Constraints on breakpoint programming mean other values are reserved under some conditions. For more information, including the effect of programming this field to a reserved value, see [Reserved DBGBCR<n>_EL1.BT values on page B8-70](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of [DBGBCR<n>_EL1.E](#) is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see “Reserved [DBGBCR<n>_EL1](#).{SSC,HMC,PMC} values” in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state. In an AArch64-only implementation, this field is reserved, RES1.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0011	DBGBVR<n>_EL1	Use for T32 instructions.
1100	DBGBVR<n>_EL1+2	Use for T32 instructions.
1111	DBGBVR<n>_EL1	Use for A64 and A32 instructions.

All other values are reserved. For more information, see “Reserved DBGBCR<n>_EL1.BAS values” in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

For more information on using the BAS field in address match breakpoints, see [Using the BAS field in Address Match breakpoints on page C6-700](#).

For Context matching breakpoints, this field is RES1 and ignored.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

E, bit [0]

Enable breakpoint DBGBVR<n>_EL1. Possible values are:

- 0 Breakpoint disabled.
- 1 Breakpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the DBGBCR<n>_EL1

This register can be read using MRS with the following syntax:

```
MRS <Xt>, <systemreg>
```

This register can be written using MSR (register) with the following syntax:

```
MSR <systemreg>, <Xt>
```

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
DBGBCR<n>_EL1	10	000	0000	n<3:0>	101

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
DBGBCR<n>_EL1	x	x	0	-	RW	n/a	RW
DBGBCR<n>_EL1	x	0	1	-	RW	RW	RW
DBGBCR<n>_EL1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If **EDSCR.TDA**==1, and **OSLSR_EL1.OSLK**==0, accesses to this register from EL1, EL2, and EL3 are trapped to Debug state.

When EL2 is implemented and is using AArch64 and **SCR_EL3.NS** == 1:

- If **MDCR_EL2.TDA**==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If **MDCR_EL3.TDA**==1, accesses to this register from EL1 and EL2 are trapped to EL3.

B12.3.2 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register [DBGBCR<n>_EL1](#).

Configurations

AArch64 System register DBGBVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register [DBGBVR<n>](#).

AArch64 System register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register [DBG BXVR<n>](#).

AArch64 System register DBGBVR<n>_EL1 is architecturally mapped to External register [DBGBVR<n>_EL1](#).

If breakpoint n is not implemented then this register is unallocated.

This register is in the Cold reset domain. On a Cold reset RW fields in this register reset to architecturally UNKNOWN values. The register is not affected by a Warm reset.

Attributes

How this register is interpreted depends on the value of [DBGBCR<n>_EL1.BT](#).

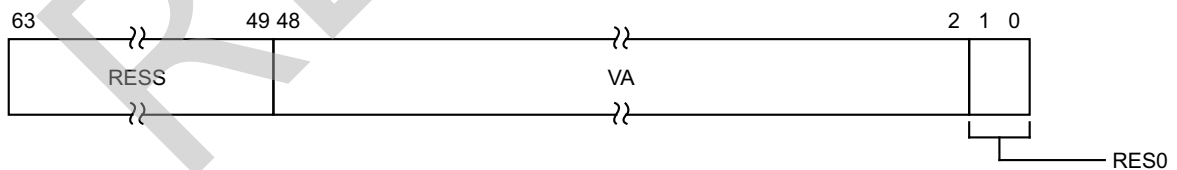
- When [DBGBCR<n>_EL1.BT](#) is 0b000x, this register holds a virtual address.
- When [DBGBCR<n>_EL1.BT](#) is 0b001x, 0b011x, or 0b110x, this register holds a Context ID.
- When [DBGBCR<n>_EL1.BT](#) is 0b100x, this register holds a VMID.
- When [DBGBCR<n>_EL1.BT](#) is 0b101x, this register holds a VMID and a Context ID.
- When [DBGBCR<n>_EL1.BT](#) is 0b111x, this register holds two Context ID values.

For other values of [DBGBCR<n>_EL1.BT](#), this register is RES0.

Field descriptions

The DBGBVR<n>_EL1 bit assignments are:

When [DBGBCR<n>_EL1.BT](#)==0b000x:



RESS, bits [63:49]

Reserved, Sign extended. Software must treat this field as RES0 if bit[48] is 0 or RES0, and as RES1 if bit[48] is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of bit [48], meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

VA, bits [48:2]

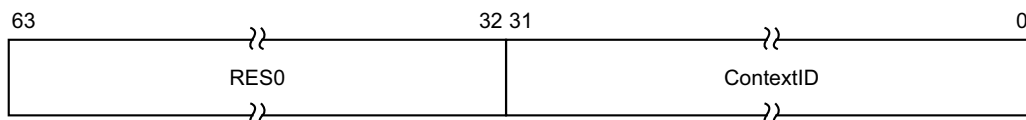
Bits[48:2] of the address value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [1:0]

Reserved, RES0.

When $DBGBCR<n>_EL1.BT == 0b001x$:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against [CONTEXTIDR_EL1](#) in the following cases:

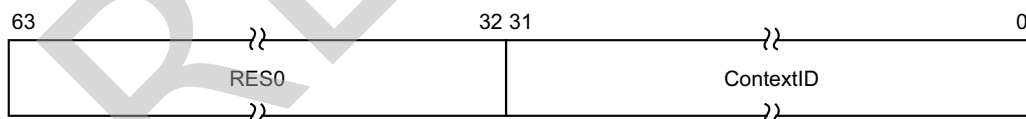
- The PE is in Secure state.
- In ARMv8.0.
- In ARMv8.1, when [HCR_EL2.E2H](#) is 0 and the PE is in Non-secure EL0, EL1 or EL2.
- In ARMv8.1, when [HCR_EL2.{E2H, TGE}](#) is {1, 0} and the PE is in Non-secure EL0 or EL1.

In ARMv8.1, when [HCR_EL2.E2H](#) is 1, the value is compared against [CONTEXTIDR_EL2](#) in the following cases:

- The PE is executing at EL2.
- [HCR_EL2.TGE](#) is 1 and the PE is in Non-secure EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>_EL1.BT == 0b011x$:



Bits [63:32]

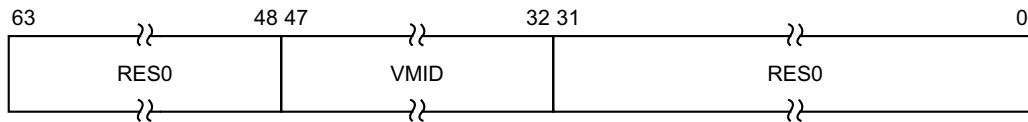
Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL1](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>_EL1.BT==0b100x$ and EL2 implemented:



Bits [63:48]

Reserved, RES0.

VMID, bits [47:32] (In ARMv8.1)

VMID value for comparison.

The VMID is 8 bits in the following cases.

- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.

In ARMv8.1 when EL2 is using AArch64, it is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

The upper 8 bits of this field are RES0 if any of the following apply:

- The implementation has an 8 bit VMID.
- [VTCR_EL2.VS](#) is 0.
- EL2 is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [47:40] (In ARMv8.0)

Reserved, RES0.

VMID, bits [39:32] (In ARMv8.0)

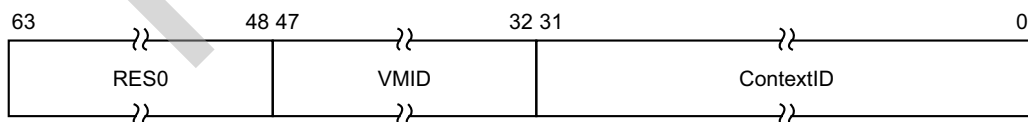
VMID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [31:0]

Reserved, RES0.

When $DBGBCR<n>_EL1.BT==0b101x$ and EL2 implemented:



Bits [63:48]

Reserved, RES0.

VMID, bits [47:32] (In ARMv8.1)

VMID value for comparison.

The VMID is 8 bits in the following cases.

- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.

In ARMv8.1 when EL2 is using AArch64, it is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

The upper 8 bits of this field are RES0 if any of the following apply:

- The implementation has an 8 bit VMID.
- [VTCR_EL2.VS](#) is 0.
- EL2 is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [47:40] (In ARMv8.0)

Reserved, RES0.

VMID, bits [39:32] (In ARMv8.0)

VMID value for comparison.

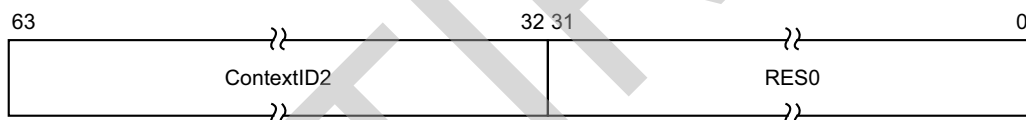
When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL1](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>_EL1.BT == 0b110x$ and EL2 implemented:



ContextID2, bits [63:32]

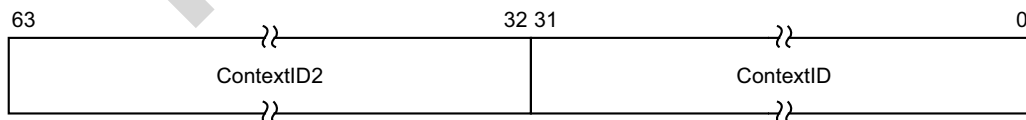
Context ID value for comparison against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [31:0]

Reserved, RES0.

When $DBGBCR<n>_EL1.BT == 0b111x$ and EL2 implemented:



ContextID2, bits [63:32]

Context ID value for comparison against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL1](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the DBGBVR<n>_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
DBGBVR<n>_EL1	10	000	0000	n<3:0>	100

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
DBGBVR<n>_EL1	x	x	0	-	RW	n/a	RW
DBGBVR<n>_EL1	x	0	1	-	RW	RW	RW
DBGBVR<n>_EL1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If **EDSCR.TDA**==1, and **OSLSR_EL1.OSLK**==0, accesses to this register from EL1, EL2, and EL3 are trapped to Debug state.

When EL2 is implemented and is using AArch64 and **SCR_EL3.NS** == 1:

- If **MDCR_EL2.TDA**==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If **MDCR_EL3.TDA**==1, accesses to this register from EL1 and EL2 are trapped to EL3.

B12.3.3 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

Purpose

Holds the saved process state on entry to Debug state.

Configurations

AArch64 System register DSPSR_EL0 is architecturally mapped to AArch32 System register [DSPSR](#).

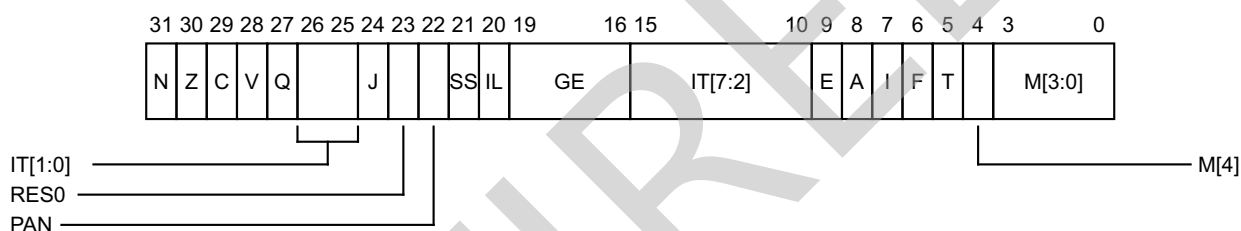
Attributes

DSPSR_EL0 is a 32-bit register.

Field descriptions

The DSPSR_EL0 bit assignments are:

When exiting Debug state to AArch32:



N, bit [31]

Copied to [CPSR.N](#) on exiting Debug state.

Z, bit [30]

Copied to [CPSR.Z](#) on exiting Debug state.

C, bit [29]

Copied to [CPSR.C](#) on exiting Debug state.

V, bit [28]

Copied to [CPSR.V](#) on exiting Debug state.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of **CPSR.PAN** on entering Debug state, and copied to **CPSR.PAN** on exiting Debug state.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of **PSTATE.SS** immediately before Debug state was entered.

IL, bit [20]

Illegal Execution state bit. Shows the value of **PSTATE.IL** immediately before Debug state was entered.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the **SCTLR.EE** bit is defined by a configuration input signal, that value also applies to the **CPSR.E** bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.

1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the Debug state entry was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that Debug state was entered from. Possible values of this bit are:

- 1 Exception taken from AArch32.

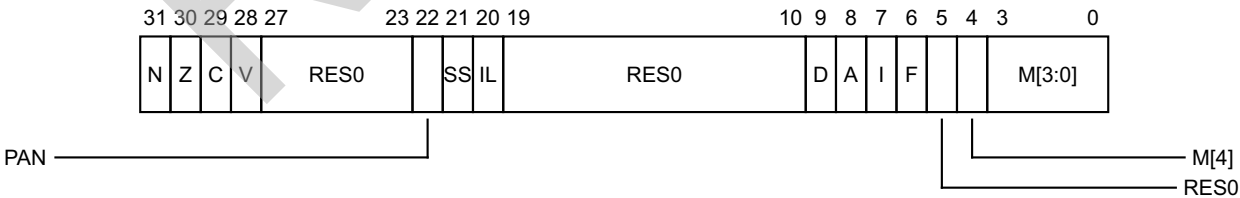
M[3:0], bits [3:0]

AArch32 mode that Debug state was entered from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

When entering Debug state from AArch64 and exiting Debug state to AArch64:



N, bit [31]

Set to the value of the N condition flag on entering Debug state, and copied to the N condition flag on exiting Debug state.

Z, bit [30]

Set to the value of the Z condition flag on entering Debug state, and copied to the Z condition flag on exiting Debug state.

C, bit [29]

Set to the value of the C condition flag on entering Debug state, and copied to the C condition flag on exiting Debug state.

V, bit [28]

Set to the value of the V condition flag on entering Debug state, and copied to the V condition flag on exiting Debug state.

Bits [27:23]

Reserved, RES0.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of PSTATE.SS immediately before Debug state was entered.

IL, bit [20]

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before Debug state was entered.

Bits [19:10]

Reserved, RES0.

D, bit [9]

Process state D mask. The possible values of this bit are:

- 0 Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
- 1 Watchpoint, Breakpoint, and Software step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

A, bit [8]

SError (System Error) mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state that Debug state was entered from. Possible values of this bit are:

- 0 Exception taken from AArch64.

M[3:0], bits [3:0]

AArch64 mode that Debug state was entered from. The possible values are:

M[3:0]	Mode
0b0000	EL0t
0b0100	EL1t
0b0101	EL1h
0b1000	EL2t
0b1001	EL2h
0b1100	EL3t
0b1101	EL3h

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
 - 0 means the SP is always SP0.
 - 1 means the exception SP is determined by the EL.

Accessing the DSPSR_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
DSPSR_EL0	11	011	0100	0101	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
DSPSR_EL0	x	x	0	RW	RW	n/a	RW
DSPSR_EL0	x	0	1	RW	RW	RW	RW
DSPSR_EL0	x	1	1	RW	n/a	RW	RW

Access to this register is from Debug state only. During normal execution this register is unallocated.

RETIRED

B12.3.4 MDCR_EL2, Monitor Debug Configuration Register (EL2)

The MDCR_EL2 characteristics are:

Purpose

Provides EL2 configuration options for self-hosted debug and the Performance Monitors Extension.

Configurations

AArch64 System register MDCR_EL2 is architecturally mapped to AArch32 System register [HDCR](#).

If EL2 is not implemented, this register is RES0 from EL3.

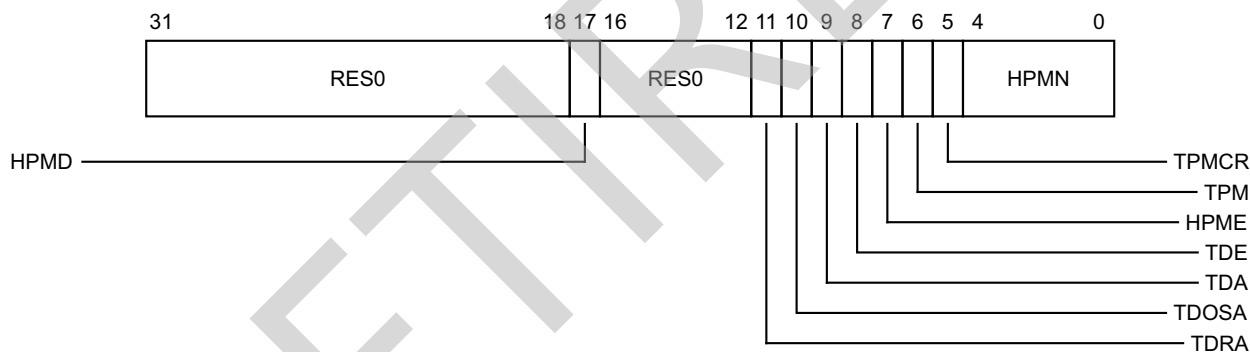
This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset these apply only if the PE resets into an Exception level that is using AArch64. Otherwise, on a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

MDCR_EL2 is a 32-bit register.

Field descriptions

The MDCR_EL2 bit assignments are:



Bits [31:18]

Reserved, RES0.

HPMD, bit [17] (In ARMv8.1)

Guest Performance Monitors Disable. This control prohibits event counting at EL2. Permitted values are:

- 0 Event counting allowed at EL2.
- 1 Event counting prohibited at EL2, unless enabled by the IMPLEMENTATION DEFINED authentication interface `ExternalSecureNoninvasiveDebugEnabled()`.

This control applies only to:

- The event counters in the range [0..HPMN).
- If [PMCR_EL0.DP](#) is set to 1, [PMCCNTR_EL0](#).

The other event counters are unaffected, and when [PMCR_EL0.DP](#) is set to 0, [PMCCNTR_EL0](#) is unaffected.

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [17] (In ARMv8.0)

Reserved, RES0.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps Non-secure System register accesses to the Debug ROM registers to EL2. This trap is from:

- Non-secure EL0 using AArch32.
 - Non-secure EL1, regardless of which Execution state it is using.
- | | |
|---|--|
| 0 | Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are not trapped to EL2. |
| 1 | Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2. |

The registers for which accesses are trapped are as follows:

AArch64: MDRAR_EL1.

AArch32: DBGDRAR, DBGDSAR.

If [MDCR_EL2.TDE](#) == 1 or [HCR_EL2.TGE](#) == 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

TDOSA, bit [10]

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states:

- | | |
|---|--|
| 0 | Non-secure EL1 System register accesses to the powerdown debug registers are not trapped to EL2. |
| 1 | Non-secure EL1 System register accesses to the powerdown debug registers are trapped to EL2. |

The registers for which accesses are trapped are as follows:

AArch64: OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, and the DBGPRCR_EL1.

AArch32: DBGOSLSR, DBGOSLAR, DBGOSDLR, and the DBGPRCR.

AArch64 and AArch32: Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Note

These registers are not accessible at EL0.

If [MDCR_EL2.TDE](#) == 1 or [HCR_EL2.TGE](#) == 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

TDA, bit [9]

Trap Debug Access. Traps Non-secure EL0 and EL1 System register accesses to those debug System registers that are not trapped by either of the following:

- MDCR_EL2.TDRA.
 - MDCR_EL2.TDOSA.
- | | |
|---|--|
| 0 | Has no effect on System register accesses to the debug registers. |
| 1 | Non-secure EL0 or EL1 System register accesses to the debug registers, other than the registers trapped by MDCR_EL2.TDRA and MDCR_EL2.TDOSA, are trapped to EL2, from both Execution states. |

MDCR_EL2.TDA does not trap accesses to the DBGDTRRX_EL0, DBGDTRTX_EL0, or DBGDTR_EL0 when the PE is in Debug state.

If MDCR_EL2.TDE == 1 or HCR_EL2.TGE == 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

TDE, bit [8]

In Non-secure state, route debug exceptions from EL1 and EL0 to EL2, and trap Debug register accesses from EL1 and EL0 to EL2. The possible values of this field are:

- 0 This control has no effect on the routing of debug exceptions, and has no effect on Non-secure accesses to debug registers.
- 1 In Non-secure state:
 - Debug exceptions generated at EL1 or EL0 are routed to EL2.
 - All accesses to Debug registers that would not be UNDEFINED if the value of this field was 0 are trapped to EL2.

When HCR_EL2.TGE == 1, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

HPME, bit [7]

Hypervisor Performance Monitors Counters Enable. The possible values of this bit are:

- 0 EL2 Performance Monitors counters disabled.
- 1 EL2 Performance Monitors# counters enabled.

When the value of this bit is 1, the Performance Monitors counters that are reserved for use from EL2 or Secure state are enabled. For more information see the description of the HPMN field.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

TPM, bit [6]

Trap Performance Monitors accesses. Traps Non-secure EL0 and EL1 accesses to all Performance Monitors registers to EL2, from both Execution states:

- 0 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are not trapped to EL2.
- 1 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to EL2.

————— Note —————

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

TPMCR, bit [5]

Trap PMCR_EL0 or PMCR accesses. Traps Non-secure EL0 and EL1 accesses to the PMCR_EL0 or PMCR to EL2.

- 0 Non-secure EL0 and EL1 accesses to the PMCR_EL0 or PMCR are not trapped to EL2.
- 1 Non-secure EL0 and EL1 accesses to the PMCR_EL0 or PMCR are trapped to EL2.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

HPMN, bits [4:0]

Defines the number of Performance Monitors counters that are accessible from Non-secure EL0 and EL1 modes.

If the Performance Monitors Extension is not implemented, this field is RES0.

In Non-secure state, HPMN divides the Performance Monitors counters as follows. For counter n in Non-secure state:

- If n is in the range $0 \leq n < \text{HPMN}$, the counter is accessible from EL1 and EL2, and from EL0 if permitted by `PMUSERENR_EL0`. `PMCR_EL0.E` enables the operation of counters in this range.
- If n is in the range $\text{HPMN} \leq n < \text{PMCR_EL0.N}$, the counter is accessible only from EL2 and from Secure state. `MDCR_EL2.HPME` enables the operation of counters in this range.

If this field is set to 0, or to a value larger than `PMCR_EL0.N`, then the following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of `MDCR_EL2.HPMN` is UNKNOWN.
- Either:
 - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if `MDCR_EL2.HPMN` is set to an UNKNOWN non-zero value less than `PMCR_EL0.N`.
 - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to the value of `PMCR_EL0.N`.

Accessing the MDCR_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
MDCR_EL2	11	100	0001	0001	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
MDCR_EL2	x	x	0	-	-	n/a	RW
MDCR_EL2	x	0	1	-	-	RW	RW
MDCR_EL2	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TDA==1, accesses to this register from EL2 are trapped to EL3.

B12.3.5 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

Purpose

Main control register for the debug implementation.

Configurations

AArch64 System register MDSCR_EL1 is architecturally mapped to AArch32 System register [DBGDSCRExt](#).

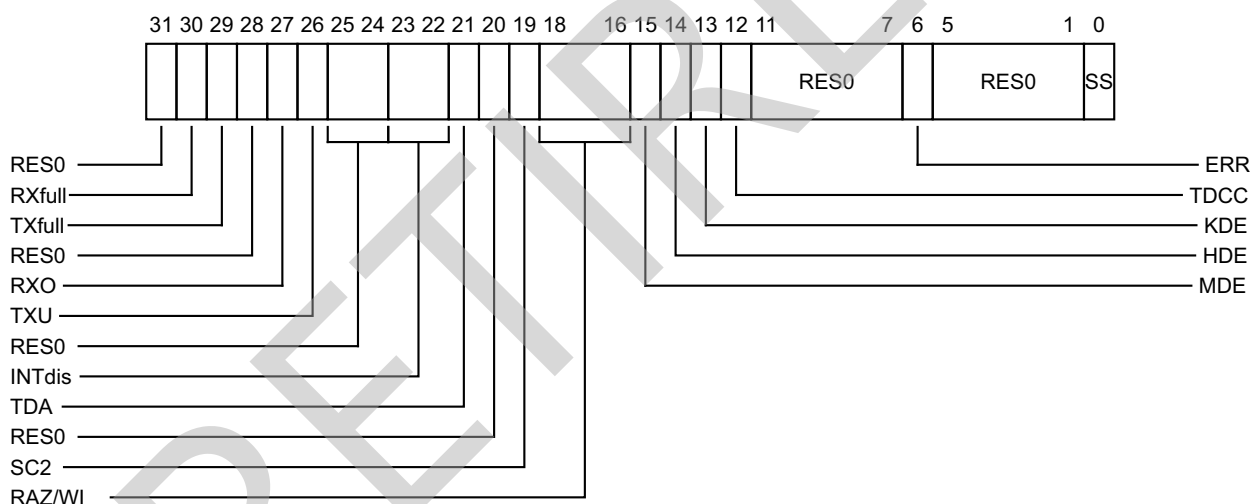
This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset these apply only if the PE resets into an Exception level that is using AArch64. Otherwise, on a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

MDSCR_EL1 is a 32-bit register.

Field descriptions

The MDSCR_EL1 bit assignments are:



Bit [31]

Reserved, RES0.

RXfull, bit [30]

Used for save/restore of [EDSCR.RXfull](#).

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.RXfull](#).

The architected behavior of this field determines the value it returns after a reset.

TXfull, bit [29]

Used for save/restore of [EDSCR.TXfull](#).

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When `OSLSR_EL1.OSLK == 1` (the OS lock is locked), this bit is RW and holds the value of `EDSCR.TXfull`.

The architected behavior of this field determines the value it returns after a reset.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of `EDSCR.RXO`.

When `OSLSR_EL1.OSLK == 0` (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When `OSLSR_EL1.OSLK == 1` (the OS lock is locked), this bit is RW and holds the value of `EDSCR.RXO`.

The architected behavior of this field determines the value it returns after a reset.

TXU, bit [26]

Used for save/restore of `EDSCR.TXU`.

When `OSLSR_EL1.OSLK == 0` (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When `OSLSR_EL1.OSLK == 1` (the OS lock is locked), this bit is RW and holds the value of `EDSCR.TXU`.

The architected behavior of this field determines the value it returns after a reset.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of `EDSCR.INTdis`.

When `OSLSR_EL1.OSLK == 0` (the OS lock is unlocked), this field is RO, and software must treat it as UNK/SBZP.

When `OSLSR_EL1.OSLK == 1` (the OS lock is locked), this field is RW and holds the value of `EDSCR.INTdis`.

The architected behavior of this field determines the value it returns after a reset.

TDA, bit [21]

Used for save/restore of `EDSCR.TDA`.

When `OSLSR_EL1.OSLK == 0` (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When `OSLSR_EL1.OSLK == 1` (the OS lock is locked), this bit is RW and holds the value of `EDSCR.TDA`.

The architected behavior of this field determines the value it returns after a reset.

Bit [20]

Reserved, RES0.

SC2, bit [19] (In ARMv8.1)

Used for save/restore of `EDSCR.SC2`.

When `OSLSR_EL1.OSLK == 0` (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When `OSLSR_EL1.OSLK == 1` (the OS lock is locked), this bit is RW and holds the value of `EDSCR.SC2`.

Bit [19] (In ARMv8.0)

Reserved, RES0.

Bits [18:16]

Reserved, RAZ/WI. Hardware must implement this as RAZ/WI. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

MDE, bit [15]

Monitor debug events. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

- 0 Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
- 1 Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

HDE, bit [14]

Used for save/restore of [EDSCR.HDE](#).

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.HDE](#).

The architected behavior of this field determines the value it returns after a reset.

KDE, bit [13]

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D. Permitted values are:

- 0 Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.
- 1 Breakpoint exceptions enabled within EL_D.

RES0 if EL_D is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

TDCC, bit [12]

Traps EL0 accesses to the DCC registers to EL1, from both Execution states:

- 0
 - EL0 using AArch64: EL0 accesses to the MDCCSR_EL0, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 registers are not trapped to EL1.
 - EL0 using AArch32: EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, [DBGDIDR](#), DBGDSAR, and DBGDRAR registers are not trapped to EL1.
- 1
 - EL0 using AArch64: EL0 accesses to the MDCCSR_EL0, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 registers are trapped to EL1.
 - EL0 using AArch32: EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, [DBGDIDR](#), DBGDSAR, and DBGDRAR registers are trapped to EL1.

Note

All accesses to these AArch32 registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

Traps of AArch32 PL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of [EDSCR.ERR](#).

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.ERR](#).

The architected behavior of this field determines the value it returns after a reset.

Bits [5:1]

Reserved, RES0.

SS, bit [0]

Software step control bit. If EL_D is using AArch64, enable Software step. Permitted values are:

0 Software step disabled

1 Software step enabled.

RES0 if EL_D is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the MDSCR_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
MDSCR_EL1	10	000	0000	0010	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
MDSCR_EL1	x	x	0	-	RW	n/a	RW
MDSCR_EL1	x	0	1	-	RW	RW	RW
MDSCR_EL1	x	1	1	-	n/a	RW	RW

Individual fields within this register might have restricted accessibility when OSLSR_EL1.OSLK == 0 (the OS lock is unlocked.) See the field descriptions for more detail.

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If MDCR_EL2.TDA==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TDA==1, accesses to this register from EL1 and EL2 are trapped to EL3.

RETIRED

B12.4 Performance Monitors registers

This section lists the ARMv8.1 Performance Monitors registers in AArch64 state, in alphabetic order.

RETIRED

B12.4.1 PMCEID0_EL0, Performance Monitors Common Event Identification register 0

The PMCEID0_EL0 characteristics are:

Purpose

Defines which common architectural and common microarchitectural feature events in the ranges 0x0000 to 0x001F and 0x4000 to 0x401F are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

Configurations

AArch64 System register PMCEID0_EL0[31:0] is architecturally mapped to AArch32 System register PMCEID0.

AArch64 System register PMCEID0_EL0[63:32] is architecturally mapped to AArch32 System register [PMCEID2](#).

AArch64 System register PMCEID0_EL0[31:0] is architecturally mapped to External register PMCEID0.

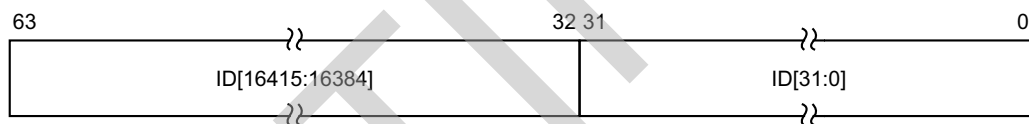
AArch64 System register PMCEID0_EL0[63:32] is architecturally mapped to External register [PMCEID2](#).

Attributes

PMCEID0_EL0 is a 64-bit register.

Field descriptions

The PMCEID0_EL0 bit assignments are:



ID[16415:16384], bits [63:32] (In ARMv8.1)

PMCEID0_EL0[63:32] maps to common events 0x4000 to 0x401F. For a list of event numbers and descriptions, see [Events, event numbers, and mnemonics on page B12-557](#).

For each bit:

- 0 The common event is not implemented.
- 1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Bits [63:32] (In ARMv8.0)

Reserved, RES0.

ID[31:0], bits [31:0]

PMCEID0_EL0[31:0] maps to common events 0x0000 to 0x001F. For a list of event numbers and descriptions, see [Events, event numbers, and mnemonics on page B12-557](#).

For each bit:

- 0 The common event is not implemented.
- 1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Accessing the PMCEID0_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
PMCEID0_EL0	11	011	1001	1100	110

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
PMCEID0_EL0	x	x	0	RO	RO	n/a	RO
PMCEID0_EL0	x	0	1	RO	RO	RO	RO
PMCEID0_EL0	x	1	1	RO	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR_EL0.EN==0, read accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If MDCR_EL2.TPM==1, Non-secure read accesses to this register from EL0 and EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TPM==1, read accesses to this register from EL0, EL1, and EL2 are trapped to EL3.

B12.4.2 PMCEID1_EL0, Performance Monitors Common Event Identification register 1

The PMCEID1_EL0 characteristics are:

Purpose

Defines which common architectural and common microarchitectural feature events in the ranges 0x0020 to 0x003F and 0x4020 to 0x403F are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

Configurations

AArch64 System register PMCEID1_EL0[31:0] is architecturally mapped to AArch32 System register PMCEID1.

AArch64 System register PMCEID1_EL0[63:32] is architecturally mapped to AArch32 System register [PMCEID3](#).

AArch64 System register PMCEID1_EL0[31:0] is architecturally mapped to External register PMCEID1.

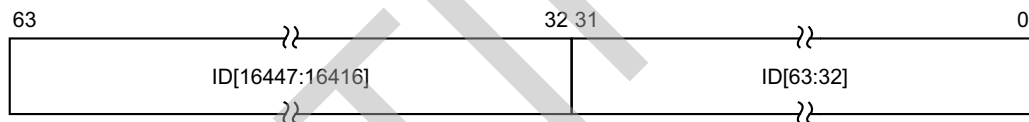
AArch64 System register PMCEID1_EL0[63:32] is architecturally mapped to External register [PMCEID3](#).

Attributes

PMCEID1_EL0 is a 64-bit register.

Field descriptions

The PMCEID1_EL0 bit assignments are:



ID[16447:16416], bits [63:32] (In ARMv8.1)

PMCEID1_EL0[63:32] maps to common events 0x4020 to 0x403F. For a list of event numbers and descriptions, see [Events, event numbers, and mnemonics on page B12-557](#).

For each bit:

- 0 The common event is not implemented.
- 1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Bits [63:32] (In ARMv8.0)

Reserved, RES0.

ID[63:32], bits [31:0]

PMCEID1_EL0[31:0] maps to common events 0x0020 to 0x003F. For a list of event numbers and descriptions, see [Events, event numbers, and mnemonics on page B12-557](#).

For each bit:

- 0 The common event is not implemented.
- 1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Accessing the PMCEID1_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
PMCEID1_EL0	11	011	1001	1100	111

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
PMCEID1_EL0	x	x	0	RO	RO	n/a	RO
PMCEID1_EL0	x	0	1	RO	RO	RO	RO
PMCEID1_EL0	x	1	1	RO	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR_EL0.EN==0, read accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If MDCR_EL2.TPM==1, Non-secure read accesses to this register from EL0 and EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TPM==1, read accesses to this register from EL0, EL1, and EL2 are trapped to EL3.

B12.4.3 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configurations

AArch64 System register PMCR_EL0 is architecturally mapped to AArch32 System register [PMCR](#).

AArch64 System register PMCR_EL0[6:0] is architecturally mapped to External register PMCR_EL0[6:0].

This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset these apply only if the PE resets into an Exception level that is using AArch64. Otherwise, on a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

PMCR_EL0 is a 32-bit register.

Field descriptions

The PMCR_EL0 bit assignments are:

31	24	23	16	15	11	10	7	6	5	4	3	2	1	0		
IMP				IDCODE				N		RES0		L	C	D	P	E

IMP, bits [31:24]

Implementer code. This field is RO with an IMPLEMENTATION DEFINED value.

The implementer codes are allocated by ARM. Values have the same interpretation as bits [31:24] of the MIDR.

IDCODE, bits [23:16]

Identification code. This field is RO with an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

N, bits [15:11]

Number of event counters. A RO field that indicates the number counters implemented. A value of 0b00000 in this field indicates that only the Cycle Count Register PMCCNTR_EL0 is implemented.

The value of this field is the number of event counters implemented. This value is in the range of 0b00000, in which case only the PMCCNTR_EL0 is implemented, to 0b11111, which indicates that the PMCCNTR_EL0 and 31 event counters are implemented.

In an implementation that includes EL2, reads of this field from Non-secure EL1 and Non-secure EL0 return the value of [MDCR_EL2.HPMN](#).

Bits [10:7]

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines which PMCCNTR_EL0 bit generates an overflow recorded by PMOVSRR[31].

- 0 Cycle counter overflow on increment that changes PMCCNTR_EL0[31] from 1 to 0.
- 1 Cycle counter overflow on increment that changes PMCCNTR_EL0[63] from 1 to 0.

ARM deprecates use of PMCR_EL0.LC = 0.

In an AArch64-only implementation, this field is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

DP, bit [5]

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

- 0 PMCCNTR_EL0, if enabled, counts when event counting is prohibited.
- 1 PMCCNTR_EL0 does not count when event counting is prohibited.

Event counting is prohibited when `ProfilingProhibited(IsSecure(), PSTATE.EL) == TRUE`.

When EL3 is not implemented, this field is RES0:

- In ARMv8.0.
- In ARMv8.1, only if EL2 is not implemented.

Otherwise this field is RW.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

X, bit [4]

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

- 0 Do not export events.
- 1 Export events where not prohibited.

This field enables the exporting of events over an event bus to another device, for example to an OPTIONAL trace macrocell. If the implementation does not include such an event bus then this field is RAZ/WI, otherwise it is an RW field.

In an implementation that includes an event bus, no events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

D, bit [3]

Clock divider. The possible values of this bit are:

- 0 When enabled, PMCCNTR_EL0 counts every clock cycle.
- 1 When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

In an AArch64-only implementation this field is RES0, otherwise it is an RW field. If PMCR_EL0.LC = 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR.D = 1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

C, bit [2]

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

- 0 No action.

1 Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Resetting PMCCNTR_EL0 does not clear the PMCCNTR_EL0 overflow bit to 0.

P, bit [1]

Event counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset all event counters accessible in the current EL, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

In Non-secure EL0 and EL1, if EL2 is implemented, a write of 1 to this bit does not reset event counters that MDCR_EL2.HPMN reserves for EL2 use.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

Resetting the event counters does not clear any overflow bits to 0.

E, bit [0]

Enable. The possible values of this bit are:

0 All counters that are accessible at Non-secure EL1, including PMCCNTR_EL0, are disabled.

1 All counters that are accessible at Non-secure EL1 are enabled by PMCNTENSET_EL0.

This bit is RW.

If EL2 is implemented, this bit does not affect the operation of event counters that MDCR_EL2.HPMN reserves for EL2 use.

When this register has an architecturally-defined reset value, this field resets to 0.

Accessing the PMCR_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
PMCR_EL0	11	011	1001	1100	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
PMCR_EL0	x	x	0	RW	RW	n/a	RW
PMCR_EL0	x	0	1	RW	RW	RW	RW
PMCR_EL0	x	1	1	RW	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR_EL0.EN==0, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If MDCR_EL2.TPM==1, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.
- If MDCR_EL2.TPMCR==1, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TPM==1, accesses to this register from EL0, EL1, and EL2 are trapped to EL3.

B12.4.4 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configurations

AArch64 System register PMEVTYPER<n>_EL0 is architecturally mapped to AArch32 System register [PMEVTYPER<n>](#).

AArch64 System register PMEVTYPER<n>_EL0 is architecturally mapped to External register PMEVTYPER<n>_EL0.

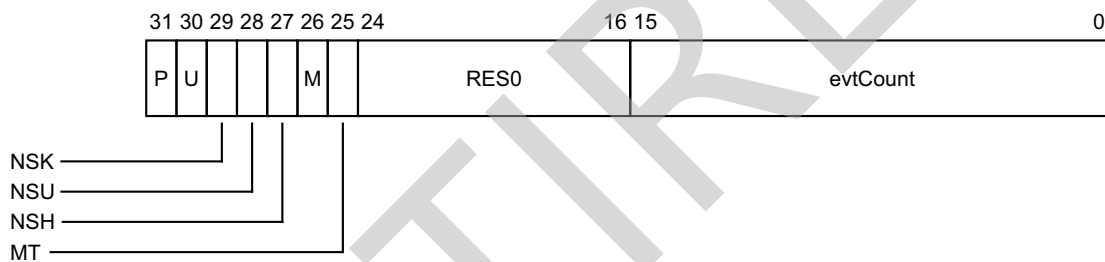
This register is in the Warm reset domain. On a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

PMEVTYPER<n>_EL0 is a 32-bit register.

Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

- 0 Count events in EL1.
- 1 Do not count events in EL1.

U, bit [30]

User filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

- 0 Count events in EL0.
- 1 Do not count events in EL0.

NSK, bit [29]

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

NSU, bit [28]

Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

NSH, bit [27]

Non-secure EL2 (Hypervisor) filtering. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

- | | |
|---|-----------------------------|
| 0 | Do not count events in EL2. |
| 1 | Count events in EL2. |

M, bit [26]

Secure EL3 filtering bit. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Secure EL3 are counted.

Otherwise, cycles in Secure EL3 are not counted.

Most applications can ignore this field and set its value to 0.

———— Note ————

This field is not visible in the AArch32 PMEVTYPER System register.

MT, bit [25]

Multithreading. When the implementation is multi-threaded, the valid values for this bit are:

- | | |
|---|--|
| 0 | Count events only on controlling PE. |
| 1 | Count events from any PE with the same affinity at level 1 and above as this PE. |

When the implementation is not multi-threaded, this bit is RES0.

———— Note ————

- An implementation is described as multi-threaded when the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. That is, the performance of PEs at the lowest affinity level is highly interdependent. On such an implementation, the value of MPIDR_EL1.MT, when read at the highest implemented Exception level, is 1.
- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

Bits [24:16]

Reserved, RES0.

evtCount, bits [15:0] (In ARMv8.1)

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event that is supported by the PE being programmed.

There are three ranges of event numbers:

- Event numbers in the range 0x000 to 0x03F are common architectural and microarchitectural events.
- Event numbers in the range 0x040 to 0x0BF are ARM recommended common architectural and microarchitectural events.
- Event numbers in the range 0x0C0 to 0x3FF are IMPLEMENTATION DEFINED events.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the event type:

- For the range 0x000 to 0x03F, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.

- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN. UNPREDICTABLE in this case means the event must not expose privileged information.

———— **Note** ————

UNPREDICTABLE means the event must not expose privileged information.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back from evtCount is UNKNOWN.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

Bits [15:10] (In ARMv8.0)

Reserved, RES0.

evtCount, bits [9:0] (In ARMv8.0)

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event that is supported by the PE being programmed.

There are three ranges of event numbers:

- Event numbers in the range 0x000 to 0x03F are common architectural and microarchitectural events.
- Event numbers in the range 0x040 to 0x0BF are ARM recommended common architectural and microarchitectural events.
- Event numbers in the range 0x0C0 to 0x3FF are IMPLEMENTATION DEFINED events.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the event type:

- For the range 0x000 to 0x03F, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN. UNPREDICTABLE in this case means the event must not expose privileged information.

———— **Note** ————

UNPREDICTABLE means the event must not expose privileged information.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back from evtCount is UNKNOWN.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

Accessing the PMEVTYPER<n>_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
PMEVTYPER<n>_EL0	11	011	1110	11:n<4:3>	n<2:0>

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
PMEVTYPER<n>_EL0	x	x	0	RW	RW	n/a	RW
PMEVTYPER<n>_EL0	x	0	1	RW	RW	RW	RW
PMEVTYPER<n>_EL0	x	1	1	RW	n/a	RW	RW

PMEVTYPER<n>_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to n.

If <n> is greater than or equal to the number of accessible counters, reads and writes of PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- For an access from Non-secure EL1, or an access from Non-secure EL0 when the value of PMUSERENR_EL0.EN is 1, if PMSELR_EL0.SEL is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2.

Note

In an implementation that includes EL2, in Non-secure state at EL0 and EL1, [MDCR_EL2.HPMN](#) identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR_EL0.EN==0, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [MDCR_EL2.TPM](#)==1, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.

When EL3 is implemented and is using AArch64:

- If [MDCR_EL3.TPM](#)==1, accesses to this register from EL0, EL1, and EL2 are trapped to EL3.

B12.5 Generic Timer registers

This section lists the ARMv8.1 Generic Timer registers in AArch64 state, in alphabetic order.

RETIRED

B12.5.1 CNTHCTL_EL2, Counter-timer Hypervisor Control register

The CNTHCTL_EL2 characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 to the physical counter and the Non-secure EL1 physical timer.

Configurations

AArch64 System register CNTHCTL_EL2 is architecturally mapped to AArch32 System register CNTHCTL.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHCTL_EL2 is a 32-bit register.

Field descriptions

The CNTHCTL_EL2 bit assignments are:

When HCR_EL2.E2H == 0:



This format applies in all ARMv8.0 implementations, and it also contains a description of the behavior when EL3 is implemented and EL2 is not implemented.

Bits [31:8]

Reserved, RES0.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTPCT_EL0 is the trigger for the event stream generated from that counter, when that stream is enabled.

EVNTDIR, bit [3]

Controls which transition of the counter register CNTPCT_EL0 trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

- 0 A 0 to 1 transition of the trigger bit triggers an event.
- 1 A 1 to 0 transition of the trigger bit triggers an event.

EVNTEN, bit [2]

Enables the generation of an event stream from the counter register CNTPCT_EL0:

- 0 Disables the event stream.
- 1 Enables the event stream.

EL1PCEN, bit [1]

Traps Non-secure EL0 and EL1 accesses to the physical timer registers to EL2.

0 From AArch64 state: Non-secure EL0 and EL1 accesses to the [CNTP_CTL_EL0](#), [CNTP_CVAL_EL0](#), and [CNTP_TVAL_EL0](#) are trapped to EL2.

From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to EL2.

1 This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

EL1PCTEN, bit [0]

Traps Non-secure EL0 and EL1 accesses to the physical counter register to EL2.

0 From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTPCT_EL0 are trapped to EL2.

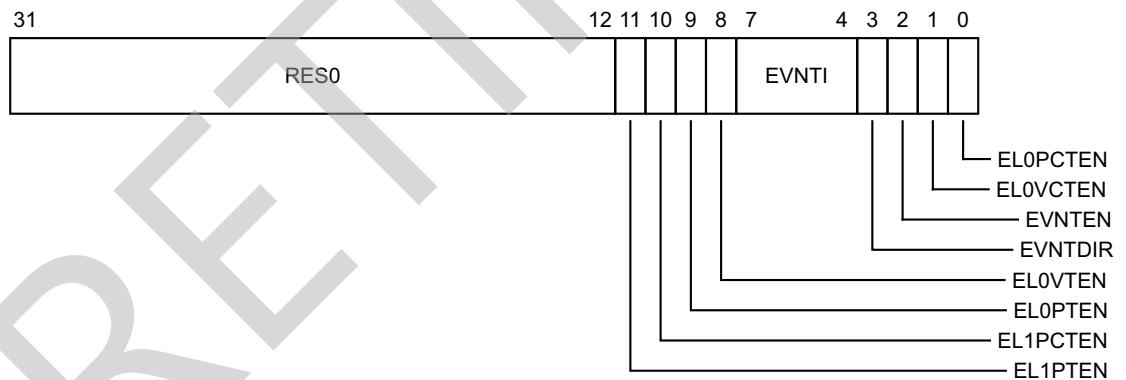
From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to EL2.

1 From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTPCT_EL0 are not trapped to EL2.

From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTPCT are not trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

When HCR_EL2.E2H == 1:



Bits [31:12]

Reserved, RES0.

EL1PTEN, bit [11]

When [HCR_EL2.TGE](#) is 0, traps Non-secure EL0 and EL1 accesses to the physical timer registers to EL2.

0 From AArch64 state: Non-secure EL0 and EL1 accesses to the [CNTP_CTL_EL0](#), [CNTP_CVAL_EL0](#), and [CNTP_TVAL_EL0](#) are trapped to EL2.

From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to EL2.

The settings in the following controls might cause a trap that is of a higher priority:

- [CNTKCTL_EL1.EL0PTEN](#)
- [CNTKCTL.PL0PTEN](#)

1 This control does not cause any instructions to be trapped.

When [HCR_EL2.TGE](#) is 1, this control does not cause any instructions to be trapped.

EL1PCTEN, bit [10]

When [HCR_EL2.TGE](#) is 0, traps Non-secure EL0 and EL1 accesses to the physical counter register to EL2.

- 0
- From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTPCT_EL0 are trapped to EL2.
- From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to EL2.
- The settings in the following controls might cause a trap that is of a higher priority:
- [CNTKCTL_EL1.EL0PCTEN](#)
 - [CNTKCTL.PL0PCTEN](#)

1 This control does not cause any instructions to be trapped.

When [HCR_EL2.TGE](#) is 1, this control does not cause any instructions to be trapped.

EL0PTEN, bit [9]

When [HCR_EL2.TGE](#) is 0, this control does not cause any instructions to be trapped.

When [HCR_EL2.TGE](#) is 1, traps EL0 accesses to the physical timer registers to EL2.

- 0
- EL0 using AArch64: EL0 accesses to the [CNTP_CTL_EL0](#), [CNTP_CVAL_EL0](#), and [CNTP_TVAL_EL0](#) registers are trapped to EL2.
- EL0 using AArch32: EL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are trapped to EL2.
- 1 This control does not cause any instructions to be trapped.

EL0VTEN, bit [8]

When [HCR_EL2.TGE](#) is 0, this control does not cause any instructions to be trapped.

When [HCR_EL2.TGE](#) is 1, traps EL0 accesses to the virtual timer registers to EL2.

- 0
- EL0 using AArch64: EL0 accesses to the [CNTV_CTL_EL0](#), [CNTV_CVAL_EL0](#), and [CNTV_TVAL_EL0](#) registers are trapped to EL2.
- EL0 using AArch32: EL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to EL2.
- 1 This control does not cause any instructions to be trapped.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register CNTPCT_EL0 is the trigger for the event stream generated from that counter, when that stream is enabled.

EVNTDIR, bit [3]

Controls which transition of the counter register CNTPCT_EL0 trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

- 0 A 0 to 1 transition of the trigger bit triggers an event.
- 1 A 1 to 0 transition of the trigger bit triggers an event.

EVNTEN, bit [2]

Enables the generation of an event stream from the counter register CNTPCT_EL0:

- 0 Disables the event stream.
- 1 Enables the event stream.

EL0VCTEN, bit [1]

When [HCR_EL2.TGE](#) is 0, this control does not cause any instructions to be trapped.

When **HCR_EL2.TGE** is 1, traps EL0 accesses to the frequency register and virtual counter register to EL2.

- 0 EL0 using AArch64: EL0 accesses to the **CNTVCT_EL0** are trapped to EL2.
EL0 using AArch64: EL0 accesses to the **CNTFRQ_EL0** register are trapped to EL2, if **CNTHCTL_EL2.EL0PCTEN** is also 0.
EL0 using AArch32: EL0 accesses to the **CNTVCT** are trapped to EL2.
EL0 using AArch32: EL0 accesses to the **CNTFRQ** register are trapped to EL2, if **CNTHCTL_EL2.EL0PCTEN** is also 0.
- 1 This control does not cause any instructions to be trapped.

EL0PCTEN, bit [0]

When **HCR_EL2.TGE** is 0, this control does not cause any instructions to be trapped.

When **HCR_EL2.TGE** is 1, traps EL0 accesses to the frequency register and physical counter register to EL2.

- 0 EL0 using AArch64: EL0 accesses to the **CNTPCT_EL0** are trapped to EL2.
EL0 using AArch64: EL0 accesses to the **CNTFRQ_EL0** register are trapped to EL2, if **CNTHCTL_EL2.EL0VCTEN** is also 0.
EL0 using AArch32: EL0 accesses to the **CNTPCT** are trapped to EL2.
EL0 using AArch32: EL0 accesses to the **CNTFRQ** and register are trapped to EL2, if **CNTHCTL_EL2.EL0VCTEN** is also 0.
- 1 This control does not cause any instructions to be trapped.

Accessing the CNTHCTL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHCTL_EL2	11	100	1110	0001	000
CNTKCTL_EL1	11	000	1110	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHCTL_EL2	x	x	0	-	-	n/a	RW
CNTHCTL_EL2	0	0	1	-	-	RW	RW
CNTHCTL_EL2	0	1	1	-	n/a	RW	RW
CNTHCTL_EL2	1	0	1	-	-	RW	RW
CNTHCTL_EL2	1	1	1	-	n/a	RW	RW

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTKCTL_EL1	x	x	0	-	CNTKCTL_EL1	n/a	CNTKCTL_EL1
CNTKCTL_EL1	0	0	1	-	CNTKCTL_EL1	CNTKCTL_EL1	CNTKCTL_EL1
CNTKCTL_EL1	0	1	1	-	n/a	CNTKCTL_EL1	CNTKCTL_EL1
CNTKCTL_EL1	1	0	1	-	CNTKCTL_EL1	RW	CNTKCTL_EL1
CNTKCTL_EL1	1	1	1	-	n/a	RW	CNTKCTL_EL1

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHCTL_EL2 or CNTKCTL_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.5.2 CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register

The CNTHP_CTL_EL2 characteristics are:

Purpose

Control register for the EL2 physical timer.

Configurations

AArch64 System register CNTHP_CTL_EL2 is architecturally mapped to AArch32 System register CNTHP_CTL.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHP_CTL_EL2 is a 32-bit register.

Field descriptions

The CNTHP_CTL_EL2 bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is asserted:

- 0 Timer condition is not asserted.
- 1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see and .

This bit is read-only.

IMASK, bit [1]

Timer output signal mask bit. Permitted values are:

- 0 Timer output signal is not masked.
- 1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

ENABLE, bit [0]

Enables the timer. Permitted values are:

- 0 Timer disabled.
- 1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTHP_TVAL_EL2](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTHP_CTL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHP_CTL_EL2	11	100	1110	0010	001
CNTP_CTL_EL0	11	011	1110	0010	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHP_CTL_EL2	x	x	0	-	-	n/a	RW
CNTHP_CTL_EL2	0	0	1	-	-	RW	RW
CNTHP_CTL_EL2	0	1	1	-	n/a	RW	RW
CNTHP_CTL_EL2	1	0	1	-	-	RW	RW
CNTHP_CTL_EL2	1	1	1	-	n/a	RW	RW
CNTP_CTL_EL0	x	x	0	CNTP_CTL_EL0	CNTP_CTL_EL0	n/a	CNTP_CTL_EL0
CNTP_CTL_EL0	0	0	1	CNTP_CTL_EL0	CNTP_CTL_EL0	CNTP_CTL_EL0	CNTP_CTL_EL0
CNTP_CTL_EL0	0	1	1	CNTP_CTL_EL0	n/a	CNTP_CTL_EL0	CNTP_CTL_EL0
CNTP_CTL_EL0	1	0	1	CNTP_CTL_EL0	CNTP_CTL_EL0	RW	RW
CNTP_CTL_EL0	1	1	1	RW	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_CTL_EL2 or CNTP_CTL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64](#) on page B12-547. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If CNTHCTL_EL2.EL0PTEN==0, Non-secure accesses to this register from EL0 are trapped to EL2.

RETIRED

B12.5.3 CNTHP_CVAL_EL2, Counter-timer Hypervisor Physical Timer CompareValue register

The CNTHP_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the EL2 physical timer.

Configurations

AArch64 System register CNTHP_CVAL_EL2 is architecturally mapped to AArch32 System register CNTHP_CVAL.

If EL2 is not implemented, this register is RES0 from EL3.

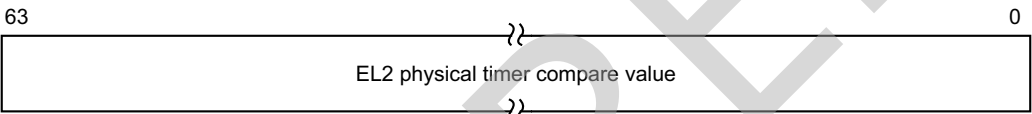
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHP_CVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHP_CVAL_EL2 bit assignments are:



Bits [63:0]

EL2 physical timer compare value.

Accessing the CNTHP_CVAL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHP_CVAL_EL2	11	100	1110	0010	010
CNTP_CVAL_EL0	11	011	1110	0010	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHP_CVAL_EL2	x	x	0	-	-	n/a	RW
CNTHP_CVAL_EL2	0	0	1	-	-	RW	RW
CNTHP_CVAL_EL2	0	1	1	-	n/a	RW	RW
CNTHP_CVAL_EL2	1	0	1	-	-	RW	RW
CNTHP_CVAL_EL2	1	1	1	-	n/a	RW	RW
CNTP_CVAL_EL0	x	x	0	CNTP_CVAL_EL0	CNTP_CVAL_EL0	n/a	CNTP_CVAL_EL0
CNTP_CVAL_EL0	0	0	1	CNTP_CVAL_EL0	CNTP_CVAL_EL0	CNTP_CVAL_EL0	CNTP_CVAL_EL0
CNTP_CVAL_EL0	0	1	1	CNTP_CVAL_EL0	n/a	CNTP_CVAL_EL0	CNTP_CVAL_EL0
CNTP_CVAL_EL0	1	0	1	CNTP_CVAL_EL0	CNTP_CVAL_EL0	RW	RW
CNTP_CVAL_EL0	1	1	1	RW	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_CVAL_EL2 or CNTP_CVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHCTL_EL2.EL0PTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.4 CNTHP_TVAL_EL2, Counter-timer Hypervisor Physical Timer TimerValue register

The CNTHP_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the EL2 physical timer.

Configurations

AArch64 System register CNTHP_TVAL_EL2 is architecturally mapped to AArch32 System register CNTHP_TVAL.

If EL2 is not implemented, this register is RES0 from EL3.

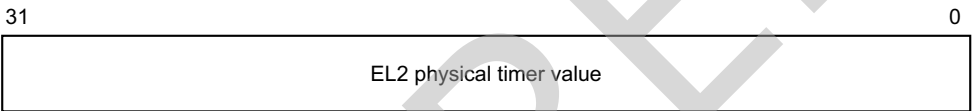
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHP_TVAL_EL2 is a 32-bit register.

Field descriptions

The CNTHP_TVAL_EL2 bit assignments are:



Bits [31:0]

EL2 physical timer value.

Accessing the CNTHP_TVAL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHP_TVAL_EL2	11	100	1110	0010	000
CNTP_TVAL_EL0	11	011	1110	0010	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHP_TVAL_EL2	x	x	0	-	-	n/a	RW
CNTHP_TVAL_EL2	0	0	1	-	-	RW	RW
CNTHP_TVAL_EL2	0	1	1	-	n/a	RW	RW
CNTHP_TVAL_EL2	1	0	1	-	-	RW	RW
CNTHP_TVAL_EL2	1	1	1	-	n/a	RW	RW
CNTP_TVAL_EL0	x	x	0	CNTP_TVAL_EL0	CNTP_TVAL_EL0	n/a	CNTP_TVAL_EL0
CNTP_TVAL_EL0	0	0	1	CNTP_TVAL_EL0	CNTP_TVAL_EL0	CNTP_TVAL_EL0	CNTP_TVAL_EL0
CNTP_TVAL_EL0	0	1	1	CNTP_TVAL_EL0	n/a	CNTP_TVAL_EL0	CNTP_TVAL_EL0
CNTP_TVAL_EL0	1	0	1	CNTP_TVAL_EL0	CNTP_TVAL_EL0	RW	RW
CNTP_TVAL_EL0	1	1	1	RW	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_TVAL_EL2 or CNTP_TVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHCTL_EL2.EL0PTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.5 CNTHV_CTL_EL2, Counter-timer Virtual Timer Control register (EL2)

The CNTHV_CTL_EL2 characteristics are:

Purpose

Control register for the EL2 virtual timer.

Configurations

AArch64 System register CNTHV_CTL_EL2 is architecturally mapped to AArch32 System register [CNTHV_CTL](#).

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHV_CTL_EL2 is a 32-bit register.

Field descriptions

The CNTHV_CTL_EL2 bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is asserted:

- 0 Timer condition is not asserted.
- 1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see [and](#) .

This bit is read-only.

IMASK, bit [1]

Timer output signal mask bit. Permitted values are:

- 0 Timer output signal is not masked.
- 1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

ENABLE, bit [0]

Enables the timer. Permitted values are:

- 0 Timer disabled.
- 1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTHV_TVAL_EL2](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTHV_CTL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHV_CTL_EL2	11	100	1110	0011	001
CNTV_CTL_EL0	11	011	1110	0011	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHV_CTL_EL2	x	x	0	-	-	n/a	RW
CNTHV_CTL_EL2	0	0	1	-	-	RW	RW
CNTHV_CTL_EL2	0	1	1	-	n/a	RW	RW
CNTHV_CTL_EL2	1	0	1	-	-	RW	RW
CNTHV_CTL_EL2	1	1	1	-	n/a	RW	RW
CNTV_CTL_EL0	x	x	0	CNTV_CTL_EL0	CNTV_CTL_EL0	n/a	CNTV_CTL_EL0
CNTV_CTL_EL0	0	0	1	CNTV_CTL_EL0	CNTV_CTL_EL0	CNTV_CTL_EL0	CNTV_CTL_EL0
CNTV_CTL_EL0	0	1	1	CNTV_CTL_EL0	n/a	CNTV_CTL_EL0	CNTV_CTL_EL0
CNTV_CTL_EL0	1	0	1	CNTV_CTL_EL0	CNTV_CTL_EL0	RW	CNTV_CTL_EL0
CNTV_CTL_EL0	1	1	1	RW	n/a	RW	CNTV_CTL_EL0

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_CTL_EL2 or CNTV_CTL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64](#) on page B12-547. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If `CNTHCTL_EL2.EL0VTEN==0`, Non-secure accesses to this register from EL0 are trapped to EL2.

RETIRED

B12.5.6 CNTHV_CVAL_EL2, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the EL2 virtual timer.

Configurations

AArch64 System register CNTHV_CVAL_EL2 is architecturally mapped to AArch32 System register [CNTHV_CVAL](#).

If EL2 is not implemented, this register is RES0 from EL3.

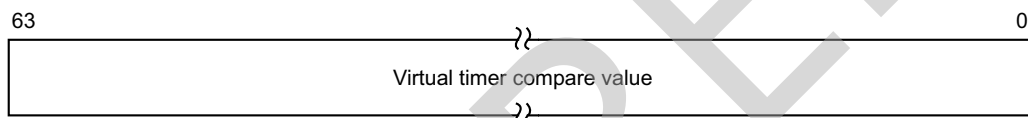
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHV_CVAL_EL2 is a 64-bit register.

Field descriptions

The CNTHV_CVAL_EL2 bit assignments are:



Bits [63:0]

Virtual timer compare value.

Accessing the CNTHV_CVAL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHV_CVAL_EL2	11	100	1110	0011	010
CNTV_CVAL_EL0	11	011	1110	0011	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHV_CVAL_EL2	x	x	0	-	-	n/a	RW
CNTHV_CVAL_EL2	0	0	1	-	-	RW	RW
CNTHV_CVAL_EL2	0	1	1	-	n/a	RW	RW
CNTHV_CVAL_EL2	1	0	1	-	-	RW	RW
CNTHV_CVAL_EL2	1	1	1	-	n/a	RW	RW
CNTV_CVAL_EL0	x	x	0	CNTV_CVAL_EL0	CNTV_CVAL_EL0	n/a	CNTV_CVAL_EL0
CNTV_CVAL_EL0	0	0	1	CNTV_CVAL_EL0	CNTV_CVAL_EL0	CNTV_CVAL_EL0	CNTV_CVAL_EL0
CNTV_CVAL_EL0	0	1	1	CNTV_CVAL_EL0	n/a	CNTV_CVAL_EL0	CNTV_CVAL_EL0
CNTV_CVAL_EL0	1	0	1	CNTV_CVAL_EL0	CNTV_CVAL_EL0	RW	CNTV_CVAL_EL0
CNTV_CVAL_EL0	1	1	1	RW	n/a	RW	CNTV_CVAL_EL0

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_CVAL_EL2 or CNTV_CVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHCTL_EL2.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.7 CNTHV_TVAL_EL2, Counter-timer Virtual Timer TimerValue register (EL2)

The CNTHV_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the EL2 virtual timer.

Configurations

AArch64 System register CNTHV_TVAL_EL2 is architecturally mapped to AArch32 System register [CNTHV_TVAL](#).

If EL2 is not implemented, this register is RES0 from EL3.

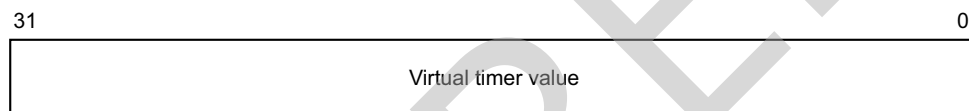
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTHV_TVAL_EL2 is a 32-bit register.

Field descriptions

The CNTHV_TVAL_EL2 bit assignments are:



Bits [31:0]

Virtual timer value.

Accessing the CNTHV_TVAL_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTHV_TVAL_EL2	11	100	1110	0011	000
CNTV_TVAL_EL0	11	011	1110	0011	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTHV_TVAL_EL2	x	x	0	-	-	n/a	RW
CNTHV_TVAL_EL2	0	0	1	-	-	RW	RW
CNTHV_TVAL_EL2	0	1	1	-	n/a	RW	RW
CNTHV_TVAL_EL2	1	0	1	-	-	RW	RW
CNTHV_TVAL_EL2	1	1	1	-	n/a	RW	RW
CNTV_TVAL_EL0	x	x	0	CNTV_TVAL_EL0	CNTV_TVAL_EL0	n/a	CNTV_TVAL_EL0
CNTV_TVAL_EL0	0	0	1	CNTV_TVAL_EL0	CNTV_TVAL_EL0	CNTV_TVAL_EL0	CNTV_TVAL_EL0
CNTV_TVAL_EL0	0	1	1	CNTV_TVAL_EL0	n/a	CNTV_TVAL_EL0	CNTV_TVAL_EL0
CNTV_TVAL_EL0	1	0	1	CNTV_TVAL_EL0	CNTV_TVAL_EL0	RW	CNTV_TVAL_EL0
CNTV_TVAL_EL0	1	1	1	RW	n/a	RW	CNTV_TVAL_EL0

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_TVAL_EL2 or CNTV_TVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHCTL_EL2.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.8 CNTKCTL_EL1, Counter-timer Kernel Control register

The CNTKCTL_EL1 characteristics are:

Purpose

In ARMv8.0, or in ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, this register controls the generation of an event stream from the virtual counter, and access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

In ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is {1, 1}, this register does not cause any event stream from the virtual counter to be generated, and does not control access to the counters and timers. The access to counters and timers at EL0 is controlled by [CNTHTCTL_EL2](#).

Configurations

AArch64 System register CNTKCTL_EL1 is architecturally mapped to AArch32 System register CNTKCTL.

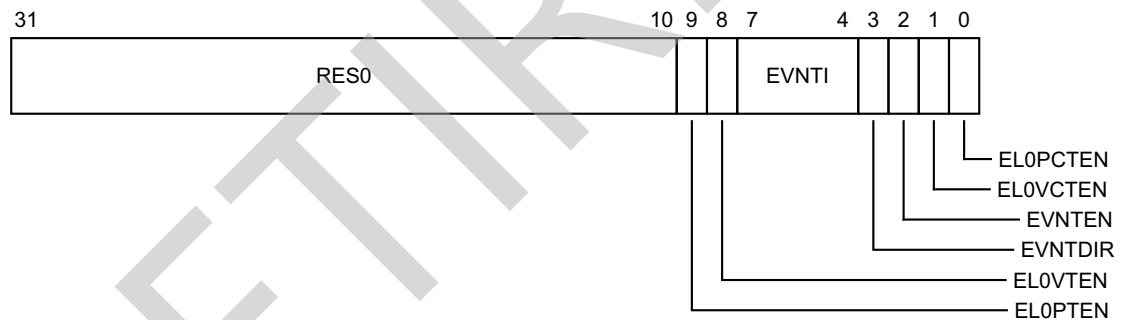
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTKCTL_EL1 is a 32-bit register.

Field descriptions

The CNTKCTL_EL1 bit assignments are:



Bits [31:10]

Reserved, RES0.

EL0PTEN, bit [9]

In ARMv8.0, or in ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, traps EL0 accesses to the physical timer registers to EL1.

0 EL0 using AArch64: EL0 accesses to the [CNTP_CTL_EL0](#), [CNTP_CVAL_EL0](#), and [CNTP_TVAL_EL0](#) registers are trapped to EL1.

EL0 using AArch32: EL0 accesses to the [CNTP_CTL](#), [CNTP_CVAL](#), and [CNTP_TVAL](#) registers are trapped to EL1.

When [HCR_EL2](#).TGE is 1, this trap is routed to EL2.

1 This control does not cause any instructions to be trapped.

In ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

EL0VTEN, bit [8]

In ARMv8.0, or in ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, traps EL0 accesses to the virtual timer registers to EL1.

0 EL0 using AArch64: EL0 accesses to the [CNTV_CTL_EL0](#), [CNTV_CVAL_EL0](#), and [CNTV_TVAL_EL0](#) registers are trapped to EL1.

EL0 using AArch32: EL0 accesses to the [CNTV_CTL](#), [CNTV_CVAL](#), and [CNTV_TVAL](#) registers are trapped to EL1.

When [HCR_EL2](#).TGE is 1, this trap is routed to EL2.

1 This control does not cause any instructions to be trapped.

In ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

EVNTI, bits [7:4]

Selects which bit (0 to 15) of the counter register [CNTVCT_EL0](#) is the trigger for the event stream generated from that counter, when that stream is enabled.

EVNTDIR, bit [3]

Controls which transition of the counter register [CNTVCT_EL0](#) trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

0 A 0 to 1 transition of the trigger bit triggers an event.

1 A 1 to 0 transition of the trigger bit triggers an event.

EVNTEN, bit [2]

In ARMv8.0, or in ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, enables the generation of an event stream from the counter register [CNTVCT_EL0](#):

0 Disables the event stream.

1 Enables the event stream.

In ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is {1, 1}, this control does not enable the event stream.

EL0VCTEN, bit [1]

In ARMv8.0, or in ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, traps EL0 accesses to the frequency register and virtual counter register to EL1.

0 EL0 using AArch64: EL0 accesses to the [CNTVCT_EL0](#) are trapped to EL1.

EL0 using AArch64: EL0 accesses to the [CNTFRQ_EL0](#) register are trapped to EL1, if [CNTKCTL_EL1](#).EL0PCTEN is also 0.

EL0 using AArch32: EL0 accesses to the [CNTVCT](#) are trapped to EL1.

EL0 using AArch32: EL0 accesses to the [CNTFRQ](#) register are trapped to EL1, if [CNTKCTL_EL1](#).EL0PCTEN is also 0.

When [HCR_EL2](#).TGE is 1, this trap is routed to EL2.

1 This control does not cause any instructions to be trapped.

In ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

EL0PCTEN, bit [0]

In ARMv8.0, or in ARMv8.1 when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, traps EL0 accesses to the frequency register and physical counter register to EL1.

0 EL0 using AArch64: EL0 accesses to the [CNTPCT_EL0](#) are trapped to EL1.

EL0 using AArch64: EL0 accesses to the [CNTFRQ_EL0](#) register are trapped to EL1, if [CNTKCTL_EL1](#).EL0VCTEN is also 0.

EL0 using AArch32: EL0 accesses to the [CNTPCT](#) are trapped to EL1.

EL0 using AArch32: EL0 accesses to the [CNTFRQ](#) and register are trapped to EL1, if [CNTKCTL_EL1](#).EL0VCTEN is also 0.

When [HCR_EL2.TGE](#) is 1, this trap is routed to EL2.

1 This control does not cause any instructions to be trapped.

In ARMv8.1 when [HCR_EL2.{E2H, TGE}](#) is {1, 1}, this control does not cause any instructions to be trapped.

Accessing the CNTKCTL_EL1

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTKCTL_EL1	11	000	1110	0001	000
CNTKCTL_EL12	11	101	1110	0001	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTKCTL_EL1	x	x	0	-	RW	n/a	RW
CNTKCTL_EL1	0	0	1	-	RW	RW	RW
CNTKCTL_EL1	0	1	1	-	n/a	RW	RW
CNTKCTL_EL1	1	0	1	-	RW	CNTHCTL_EL2	RW
CNTKCTL_EL1	1	1	1	-	n/a	CNTHCTL_EL2	RW
CNTKCTL_EL12	x	x	0	-	-	n/a	-
CNTKCTL_EL12	0	0	1	-	-	-	-
CNTKCTL_EL12	0	1	1	-	n/a	-	-
CNTKCTL_EL12	1	0	1	-	-	RW	RW
CNTKCTL_EL12	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTKCTL_EL1 or CNTKCTL_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

B12.5.9 CNTP_CTL_EL0, Counter-timer Physical Timer Control register

The CNTP_CTL_EL0 characteristics are:

Purpose

Control register for the EL1 physical timer.

Configurations

AArch64 System register CNTP_CTL_EL0 is architecturally mapped to AArch32 System register CNTP_CTL.

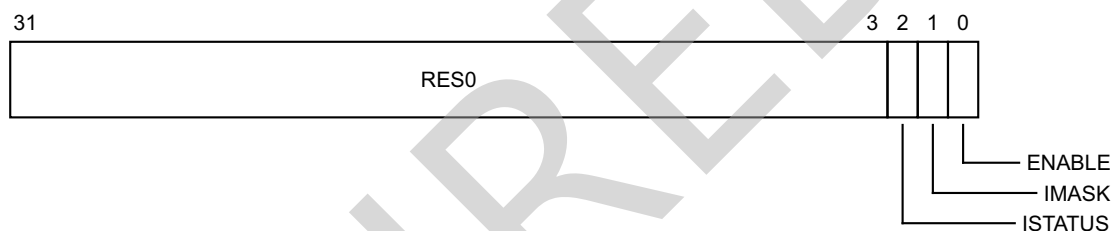
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTP_CTL_EL0 is a 32-bit register.

Field descriptions

The CNTP_CTL_EL0 bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is asserted:

- 0 Timer condition is not asserted.
- 1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see and .

This bit is read-only.

IMASK, bit [1]

Timer output signal mask bit. Permitted values are:

- 0 Timer output signal is not masked.
- 1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

ENABLE, bit [0]

Enables the timer. Permitted values are:

- 0 Timer disabled.
- 1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTP_TVAL_EL0](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTP_CTL_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTP_CTL_EL0	11	011	1110	0010	001
CNTP_CTL_EL02	11	101	1110	0010	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTP_CTL_EL0	x	x	0	RW	RW	n/a	RW
CNTP_CTL_EL0	0	0	1	RW	RW	RW	RW
CNTP_CTL_EL0	0	1	1	RW	n/a	RW	RW
CNTP_CTL_EL0	1	0	1	RW	RW	CNTHP_CTL_EL2	RW
CNTP_CTL_EL0	1	1	1	CNTHP_CTL_EL2	n/a	CNTHP_CTL_EL2	RW
CNTP_CTL_EL02	x	x	0	-	-	n/a	-
CNTP_CTL_EL02	0	0	1	-	-	-	-
CNTP_CTL_EL02	0	1	1	-	n/a	-	-
CNTP_CTL_EL02	1	0	1	-	-	RW	RW
CNTP_CTL_EL02	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_CTL_EL0 or CNTP_CTL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When $\text{HCR_EL2.E2H} = 0$:

- If $\text{CNTKCTL_EL1.EL0PTEN} = 0$, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 0)$:

- If $\text{CNTHCTL_EL2.EL1PCEN} = 0$, Non-secure accesses to this register from EL1 are trapped to EL2.
- If $\text{CNTHCTL_EL2.EL1PCEN} = 0$, and $\text{CNTKCTL_EL1.EL0PTEN} = 1$, Non-secure accesses to this register from EL0 are trapped to EL2.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 0)$:

- If $\text{CNTHCTL_EL2.EL1PTEN} = 0$, Non-secure accesses to this register from EL1 are trapped to EL2.
- If $\text{CNTHCTL_EL2.EL1PTEN} = 0$, and $\text{CNTKCTL_EL1.EL0PTEN} = 1$, Non-secure accesses to this register from EL0 are trapped to EL2.
- If $\text{CNTKCTL_EL1.EL0PTEN} = 0$, Non-secure accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 1)$:

- If $\text{CNTHCTL_EL2.EL0PTEN} = 0$, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.10 CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configurations

AArch64 System register CNTP_CVAL_EL0 is architecturally mapped to AArch32 System register CNTP_CVAL.

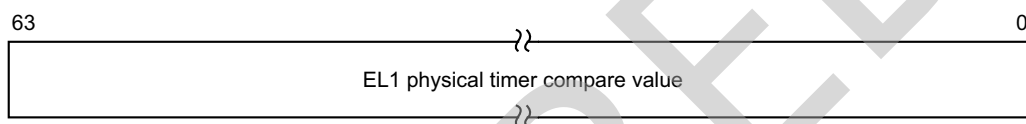
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTP_CVAL_EL0 is a 64-bit register.

Field descriptions

The CNTP_CVAL_EL0 bit assignments are:



Bits [63:0]

EL1 physical timer compare value.

Accessing the CNTP_CVAL_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTP_CVAL_EL0	11	011	1110	0010	010
CNTP_CVAL_EL02	11	101	1110	0010	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTP_CVAL_EL0	x	x	0	RW	RW	n/a	RW
CNTP_CVAL_EL0	0	0	1	RW	RW	RW	RW
CNTP_CVAL_EL0	0	1	1	RW	n/a	RW	RW
CNTP_CVAL_EL0	1	0	1	RW	RW	CNTHP_CVAL_EL2	RW
CNTP_CVAL_EL0	1	1	1	CNTHP_CVAL_EL2	n/a	CNTHP_CVAL_EL2	RW
CNTP_CVAL_EL02	x	x	0	-	-	n/a	-
CNTP_CVAL_EL02	0	0	1	-	-	-	-
CNTP_CVAL_EL02	0	1	1	-	n/a	-	-
CNTP_CVAL_EL02	1	0	1	-	-	RW	RW
CNTP_CVAL_EL02	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_CVAL_EL0 or CNTP_CVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When $\text{HCR_EL2.E2H} = 0$:

- If [CNTKCTL_EL1.EL0PTEN](#)==0, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 0)$:

- If [CNTHCTL_EL2.EL1PCEN](#)==0, Non-secure accesses to this register from EL1 are trapped to EL2.
- If [CNTHCTL_EL2.EL1PCEN](#)==0, and [CNTKCTL_EL1.EL0PTEN](#)==1, Non-secure accesses to this register from EL0 are trapped to EL2.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 0)$:

- If [CNTHCTL_EL2.EL1PTEN](#)==0, Non-secure accesses to this register from EL1 are trapped to EL2.
- If [CNTHCTL_EL2.EL1PTEN](#)==0, and [CNTKCTL_EL1.EL0PTEN](#)==1, Non-secure accesses to this register from EL0 are trapped to EL2.
- If [CNTKCTL_EL1.EL0PTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 1)$:

- If [CNTHCTL_EL2.EL0PTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTP_TVAL_EL0	x	x	0	RW	RW	n/a	RW
CNTP_TVAL_EL0	0	0	1	RW	RW	RW	RW
CNTP_TVAL_EL0	0	1	1	RW	n/a	RW	RW
CNTP_TVAL_EL0	1	0	1	RW	RW	CNTHP_TVAL_EL2	RW
CNTP_TVAL_EL0	1	1	1	CNTHP_TVAL_EL2	n/a	CNTHP_TVAL_EL2	RW
CNTP_TVAL_EL02	x	x	0	-	-	n/a	-
CNTP_TVAL_EL02	0	0	1	-	-	-	-
CNTP_TVAL_EL02	0	1	1	-	n/a	-	-
CNTP_TVAL_EL02	1	0	1	-	-	RW	RW
CNTP_TVAL_EL02	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_TVAL_EL0 or CNTP_TVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When [HCR_EL2.E2H](#) == 0:

- If [CNTKCTL_EL1.EL0PTEN](#)==0, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and ([SCR_EL3.NS](#) == 1) AND ([HCR_EL2.E2H](#) == 0):

- If [CNTHCTL_EL2.EL1PCEN](#)==0, Non-secure accesses to this register from EL1 are trapped to EL2.
- If [CNTHCTL_EL2.EL1PCEN](#)==0, and [CNTKCTL_EL1.EL0PTEN](#)==1, Non-secure accesses to this register from EL0 are trapped to EL2.

When EL2 is implemented and is using AArch64 and ([SCR_EL3.NS](#) == 1) AND ([HCR_EL2.E2H](#) == 1) AND ([HCR_EL2.TGE](#) == 0):

- If [CNTHCTL_EL2.EL1PTEN](#)==0, Non-secure accesses to this register from EL1 are trapped to EL2.
- If [CNTHCTL_EL2.EL1PTEN](#)==0, and [CNTKCTL_EL1.EL0PTEN](#)==1, Non-secure accesses to this register from EL0 are trapped to EL2.
- If [CNTKCTL_EL1.EL0PTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and ([SCR_EL3.NS](#) == 1) AND ([HCR_EL2.E2H](#) == 1) AND ([HCR_EL2.TGE](#) == 1):

- If [CNTHCTL_EL2.EL0PTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.12 CNTV_CTL_EL0, Counter-timer Virtual Timer Control register

The CNTV_CTL_EL0 characteristics are:

Purpose

Control register for the virtual timer.

Configurations

AArch64 System register CNTV_CTL_EL0 is architecturally mapped to AArch32 System register CNTV_CTL.

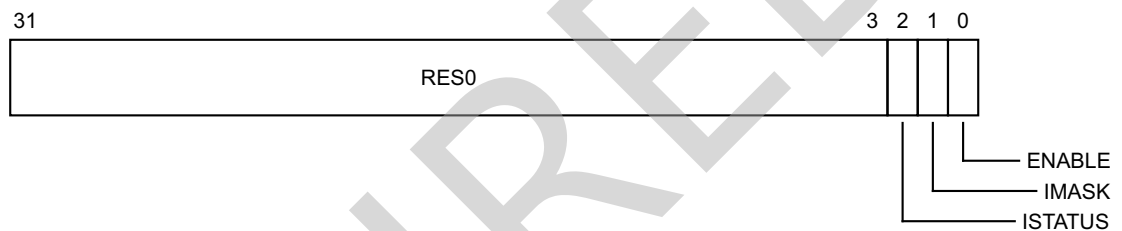
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTV_CTL_EL0 is a 32-bit register.

Field descriptions

The CNTV_CTL_EL0 bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is asserted:

- 0 Timer condition is not asserted.
- 1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see and .

This bit is read-only.

IMASK, bit [1]

Timer output signal mask bit. Permitted values are:

- 0 Timer output signal is not masked.
- 1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

ENABLE, bit [0]

Enables the timer. Permitted values are:

- 0 Timer disabled.
- 1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTV_TVAL_EL0](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTV_CTL_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTV_CTL_EL0	11	011	1110	0011	001
CNTV_CTL_EL02	11	101	1110	0011	001

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTV_CTL_EL0	x	x	0	RW	RW	n/a	RW
CNTV_CTL_EL0	0	0	1	RW	RW	RW	RW
CNTV_CTL_EL0	0	1	1	RW	n/a	RW	RW
CNTV_CTL_EL0	1	0	1	RW	RW	CNTHV_CTL_EL2	RW
CNTV_CTL_EL0	1	1	1	CNTHV_CTL_EL2	n/a	CNTHV_CTL_EL2	RW
CNTV_CTL_EL02	x	x	0	-	-	n/a	-
CNTV_CTL_EL02	0	0	1	-	-	-	-
CNTV_CTL_EL02	0	1	1	-	n/a	-	-
CNTV_CTL_EL02	1	0	1	-	-	RW	RW
CNTV_CTL_EL02	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_CTL_EL0 or CNTV_CTL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64](#) on page B12-547. Subject to the prioritization rules:

When $\text{HCR_EL2.E2H} = 0$:

- If $\text{CNTKCTL_EL1.EL0VTEN} = 0$, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 0)$:

- If $\text{CNTKCTL_EL1.EL0VTEN} = 0$, Non-secure accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 1)$:

- If $\text{CNTHCTL_EL2.EL0VTEN} = 0$, Non-secure accesses to this register from EL0 are trapped to EL2.

RETIRED

B12.5.13 CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the virtual timer.

Configurations

AArch64 System register CNTV_CVAL_EL0 is architecturally mapped to AArch32 System register CNTV_CVAL.

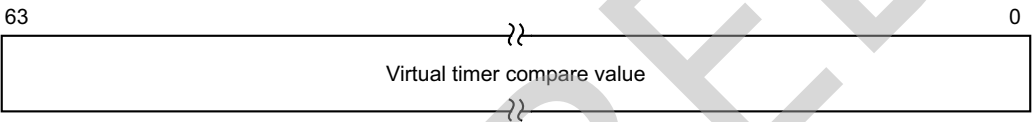
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTV_CVAL_EL0 is a 64-bit register.

Field descriptions

The CNTV_CVAL_EL0 bit assignments are:



Bits [63:0]

Virtual timer compare value.

Accessing the CNTV_CVAL_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTV_CVAL_EL0	11	011	1110	0011	010
CNTV_CVAL_EL02	11	101	1110	0011	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTV_CVAL_EL0	x	x	0	RW	RW	n/a	RW
CNTV_CVAL_EL0	0	0	1	RW	RW	RW	RW
CNTV_CVAL_EL0	0	1	1	RW	n/a	RW	RW
CNTV_CVAL_EL0	1	0	1	RW	RW	CNTHV_CVAL_EL2	RW
CNTV_CVAL_EL0	1	1	1	CNTHV_CVAL_EL2	n/a	CNTHV_CVAL_EL2	RW
CNTV_CVAL_EL02	x	x	0	-	-	n/a	-
CNTV_CVAL_EL02	0	0	1	-	-	-	-
CNTV_CVAL_EL02	0	1	1	-	n/a	-	-
CNTV_CVAL_EL02	1	0	1	-	-	RW	RW
CNTV_CVAL_EL02	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_CVAL_EL0 or CNTV_CVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When [HCR_EL2.E2H](#) == 0:

- If [CNTKCTL_EL1.EL0VTEN](#)==0, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and ([SCR_EL3.NS](#) == 1) AND ([HCR_EL2.E2H](#) == 1) AND ([HCR_EL2.TGE](#) == 0):

- If [CNTKCTL_EL1.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and ([SCR_EL3.NS](#) == 1) AND ([HCR_EL2.E2H](#) == 1) AND ([HCR_EL2.TGE](#) == 1):

- If [CNTHCTL_EL2.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.14 CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configurations

AArch64 System register CNTV_TVAL_EL0 is architecturally mapped to AArch32 System register CNTV_TVAL.

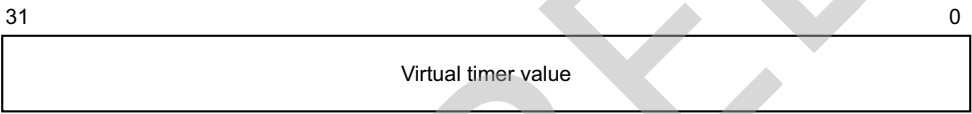
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTV_TVAL_EL0 is a 32-bit register.

Field descriptions

The CNTV_TVAL_EL0 bit assignments are:



Bits [31:0]

Virtual timer value.

Accessing the CNTV_TVAL_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTV_TVAL_EL0	11	011	1110	0011	000
CNTV_TVAL_EL02	11	101	1110	0011	000

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTV_TVAL_EL0	x	x	0	RW	RW	n/a	RW
CNTV_TVAL_EL0	0	0	1	RW	RW	RW	RW
CNTV_TVAL_EL0	0	1	1	RW	n/a	RW	RW
CNTV_TVAL_EL0	1	0	1	RW	RW	CNTHV_TVAL_EL2	RW
CNTV_TVAL_EL0	1	1	1	CNTHV_TVAL_EL2	n/a	CNTHV_TVAL_EL2	RW
CNTV_TVAL_EL02	x	x	0	-	-	n/a	-
CNTV_TVAL_EL02	0	0	1	-	-	-	-
CNTV_TVAL_EL02	0	1	1	-	n/a	-	-
CNTV_TVAL_EL02	1	0	1	-	-	RW	RW
CNTV_TVAL_EL02	1	1	1	-	n/a	RW	RW

When [HCR_EL2.E2H](#) is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_TVAL_EL0 or CNTV_TVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When $\text{HCR_EL2.E2H} = 0$:

- If $\text{CNTKCTL_EL1.EL0VTEN} = 0$, accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 0)$:

- If $\text{CNTKCTL_EL1.EL0VTEN} = 0$, Non-secure accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 1)$:

- If $\text{CNTHCTL_EL2.EL0VTEN} = 0$, Non-secure accesses to this register from EL0 are trapped to EL2.

B12.5.15 CNTVCT_EL0, Counter-timer Virtual Count register

The CNTVCT_EL0 characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT_EL0 minus the virtual offset visible in [CNTVOFF_EL2](#).

Configurations

AArch64 System register CNTVCT_EL0 is architecturally mapped to AArch32 System register [CNTVCT](#).

The value of this register is the same as the value of CNTPCT_EL0 in the following conditions:

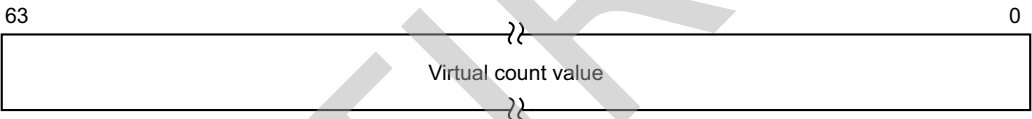
- When EL2 is not implemented.
- When EL2 is implemented, [HCR_EL2.E2H](#) is 1, and this register is read from EL2.
- When EL2 is implemented, [HCR_EL2](#).{E2H, TGE} is {1, 1}, and this register is read from Non-secure EL0 or EL2.

Attributes

CNTVCT_EL0 is a 64-bit register.

Field descriptions

The CNTVCT_EL0 bit assignments are:



Bits [63:0]

Virtual count value.

Accessing the CNTVCT_EL0

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTVCT_EL0	11	011	1110	0000	010

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTVCT_EL0	x	x	0	RO	RO	n/a	RO
CNTVCT_EL0	x	0	1	RO	RO	RO	RO
CNTVCT_EL0	x	1	1	RO	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When $\text{HCR_EL2.E2H} = 0$:

- If $\text{CNTKCTL_EL1.EL0VCTEN} = 0$, read accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 0)$:

- If $\text{CNTKCTL_EL1.EL0VCTEN} = 0$, Non-secure read accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and $(\text{SCR_EL3.NS} = 1)$ AND $(\text{HCR_EL2.E2H} = 1)$ AND $(\text{HCR_EL2.TGE} = 1)$:

- If $\text{CNTHCTL_EL2.EL0PCTEN} = 0$, Non-secure read accesses to this register from EL0 are trapped to EL2.

B12.5.16 CNTVOFF_EL2, Counter-timer Virtual Offset register

The CNTVOFF_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT_EL0 and the virtual count value visible in CNTVCT_EL0.

Configurations

AArch64 System register CNTVOFF_EL2 is architecturally mapped to AArch32 System register CNTVOFF.

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

Note

When EL2 is implemented and is using AArch64, the virtual counter uses a fixed virtual offset of zero in the following situations:

- HCR_EL2.E2H is 1, and CNTVCT_EL0 is read from EL2.
- HCR_EL2.{E2H, TGE} is {1, 1}, and either:
 - CNTVCT_EL0 is read from Non-secure EL0 or EL2.
 - CNTVCT is read from Non-secure EL0.

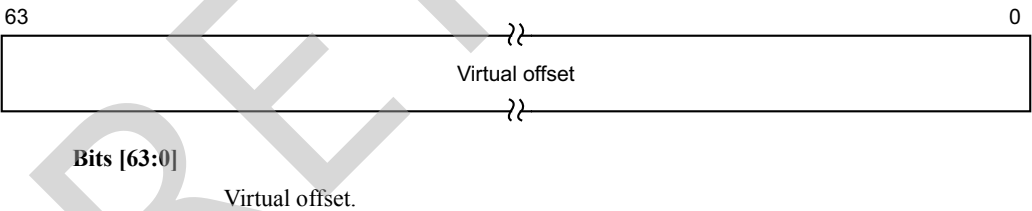
RW fields in this register reset to architecturally UNKNOWN values.

Attributes

CNTVOFF_EL2 is a 64-bit register.

Field descriptions

The CNTVOFF_EL2 bit assignments are:



Accessing the CNTVOFF_EL2

This register can be read using MRS with the following syntax:

MRS <Xt>, <systemreg>

This register can be written using MSR (register) with the following syntax:

MSR <systemreg>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<systemreg>	op0	op1	CRn	CRm	op2
CNTVOFF_EL2	11	100	1110	0000	011

Accessibility

The register is accessible in software as follows:

<systemreg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
CNTVOFF_EL2	x	x	0	-	-	n/a	RW
CNTVOFF_EL2	x	0	1	-	-	RW	RW
CNTVOFF_EL2	x	1	1	-	n/a	RW	RW

RETIRED

B12.6 System instructions

This section lists the ARMv8.1 System instructions in AArch64 state, in alphabetic order.

RETIRED

B12.6.1 AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

Purpose

Performs stage 1 and 2 address translations, with permissions as if reading from the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

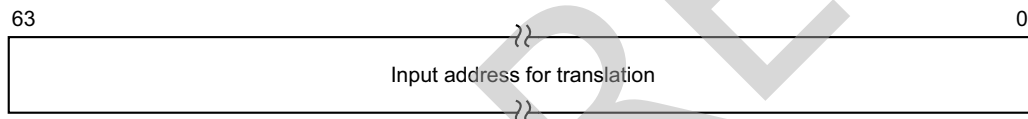
There are no configuration notes.

Attributes

AT S12E0R is a 64-bit system operation.

Field descriptions

The AT S12E0R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E0R instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S12E0R	100	0111	1000	110

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E0R	x	x	0	-	-	n/a	WO
S12E0R	0	0	1	-	-	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E0R	0	1	1	-	n/a	WO	WO
S12E0R	1	0	1	-	-	WO	WO
S12E0R	1	1	1	-	n/a	WO	WO

If EL2 is not implemented, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of [SCR_EL3.NS](#) is 0, this instruction executes as [AT S1E0R](#).

RETIRED

B12.6.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

Purpose

Performs stage 1 and 2 address translations, with permissions as if writing to the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

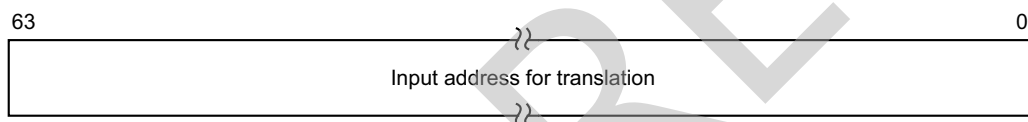
There are no configuration notes.

Attributes

AT S12E0W is a 64-bit system operation.

Field descriptions

The AT S12E0W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E0W instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S12E0W	100	0111	1000	111

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E0W	x	x	0	-	-	n/a	WO
S12E0W	0	0	1	-	-	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E0W	0	1	1	-	n/a	WO	WO
S12E0W	1	0	1	-	-	WO	WO
S12E0W	1	1	1	-	n/a	WO	WO

If EL2 is not implemented, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of [SCR_EL3.NS](#) is 0, this instruction executes as [AT S1E0W](#).

RETIRED

B12.6.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

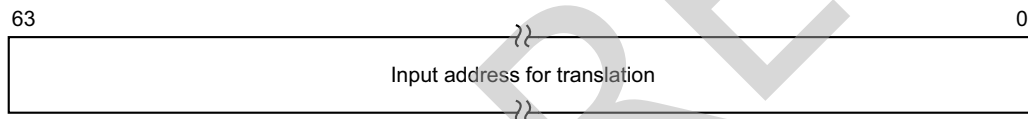
There are no configuration notes.

Attributes

AT S12E1R is a 64-bit system operation.

Field descriptions

The AT S12E1R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E1R instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S12E1R	100	0111	1000	100

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E1R	x	x	0	-	-	n/a	WO
S12E1R	0	0	1	-	-	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E1R	0	1	1	-	n/a	WO	WO
S12E1R	1	0	1	-	-	WO	WO
S12E1R	1	1	1	-	n/a	WO	WO

If EL2 is not implemented, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of [SCR_EL3.NS](#) is 0, this instruction executes as [AT S1E1R](#).

RETIRED

B12.6.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

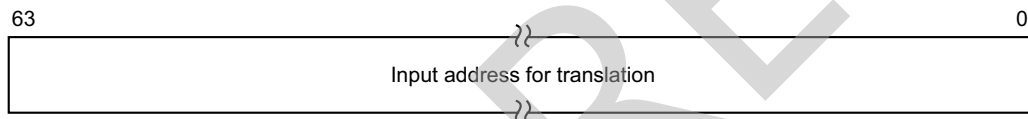
There are no configuration notes.

Attributes

AT S12E1W is a 64-bit system operation.

Field descriptions

The AT S12E1W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S12E1W instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S12E1W	100	0111	1000	101

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E1W	x	x	0	-	-	n/a	WO
S12E1W	0	0	1	-	-	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S12E1W	0	1	1	-	n/a	WO	WO
S12E1W	1	0	1	-	-	WO	WO
S12E1W	1	1	1	-	n/a	WO	WO

If EL2 is not implemented, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of [SCR_EL3.NS](#) is 0, this instruction executes as [AT S1E1W](#).

RETIRED

B12.6.5 AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

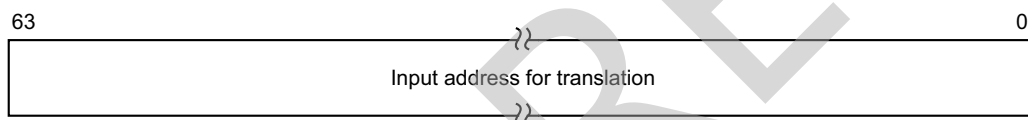
There are no configuration notes.

Attributes

AT S1E0R is a 64-bit system operation.

Field descriptions

The AT S1E0R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E0R instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S1E0R	000	0111	1000	010

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E0R	x	x	0	-	WO	n/a	WO
S1E0R	0	0	1	-	WO	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E0R	0	1	1	-	n/a	WO	WO
S1E0R	1	0	1	-	WO	WO	WO
S1E0R	1	1	1	-	n/a	WO	WO

RETIRED

B12.6.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

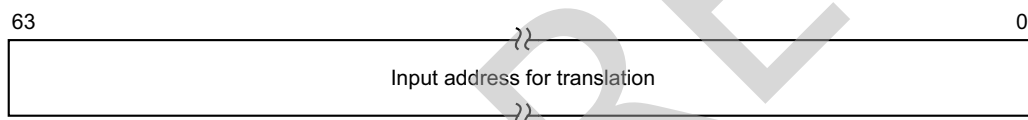
There are no configuration notes.

Attributes

AT S1E0W is a 64-bit system operation.

Field descriptions

The AT S1E0W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E0W instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S1E0W	000	0111	1000	011

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E0W	x	x	0	-	WO	n/a	WO
S1E0W	0	0	1	-	WO	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E0W	0	1	1	-	n/a	WO	WO
S1E0W	1	0	1	-	WO	WO	WO
S1E0W	1	1	1	-	n/a	WO	WO

RETIRED

B12.6.7 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

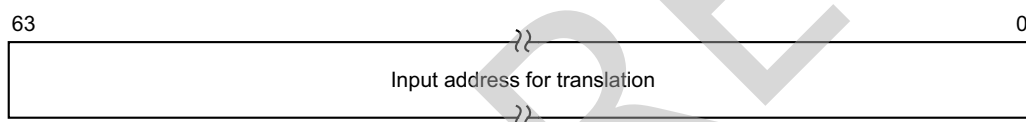
There are no configuration notes.

Attributes

AT S1E1R is a 64-bit system operation.

Field descriptions

The AT S1E1R input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E1R instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S1E1R	000	0111	1000	000

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E1R	x	x	0	-	WO	n/a	WO
S1E1R	0	0	1	-	WO	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E1R	0	1	1	-	n/a	WO	WO
S1E1R	1	0	1	-	WO	WO	WO
S1E1R	1	1	1	-	n/a	WO	WO

RETIRED

B12.6.8 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address, using the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

Configurations

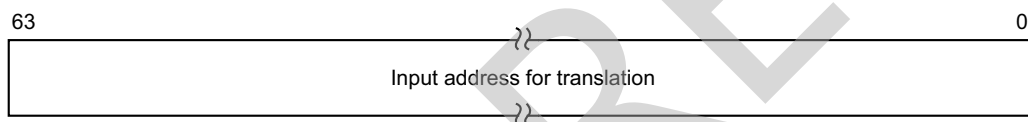
There are no configuration notes.

Attributes

AT S1E1W is a 64-bit system operation.

Field descriptions

The AT S1E1W input value bit assignments are:



Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing the AT S1E1W instruction

This instruction can be executed using AT with the following syntax:

AT <at_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<at_op>	op1	CRn	CRm	op2
S1E1W	000	0111	1000	001

Accessibility

The instruction is executable as follows:

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E1W	x	x	0	-	WO	n/a	WO
S1E1W	0	0	1	-	WO	WO	WO

<at_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
S1E1W	0	1	1	-	n/a	WO	WO
S1E1W	1	0	1	-	WO	WO	WO
S1E1W	1	1	1	-	n/a	WO	WO

RETIRED

B12.6.9 TLBI ASIDE1, TLB Invalidate by ASID, EL1

The TLBI ASIDE1 characteristics are:

Purpose

Invalidate stage 1 TLB entries for the given ASID and, if applicable, the current VMID, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [ASID](#).

Configurations

There are no configuration notes.

Attributes

TLBI ASIDE1 is a 64-bit system operation.

Field descriptions

The TLBI ASIDE1 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this operation.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

Bits [47:0]

Reserved, RES0.

Executing the TLBI ASIDE1 instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
ASIDE1	000	1000	0111	010

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ASIDE1	x	x	0	-	WO	n/a	WO
ASIDE1	0	0	1	-	WO	WO	WO
ASIDE1	0	1	1	-	n/a	WO	WO
ASIDE1	1	0	1	-	WO	WO	WO
ASIDE1	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TTLB==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TTLB==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.10 TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS characteristics are:

Purpose

Invalidate stage 1 TLB entries for the given ASID and, if applicable, the current VMID on all PEs in the same Inner Shareable domain, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [ASID](#).

Configurations

There are no configuration notes.

Attributes

TLBI ASIDE1IS is a 64-bit system operation.

Field descriptions

The TLBI ASIDE1IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this operation.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

Bits [47:0]

Reserved, RES0.

Executing the TLBI ASIDE1IS instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
ASIDE1IS	000	1000	0011	010

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
ASIDE1IS	x	x	0	-	WO	n/a	WO
ASIDE1IS	0	0	1	-	WO	WO	WO
ASIDE1IS	0	1	1	-	n/a	WO	WO
ASIDE1IS	1	0	1	-	WO	WO	WO
ASIDE1IS	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.11 TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1 characteristics are:

Purpose

Invalidate stage 1 TLB entries for the given VA and, if applicable, the current VMID, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VAA](#).

Configurations

There are no configuration notes.

Attributes

TLBI VAAE1 is a 64-bit system operation.

Field descriptions

The TLBI VAAE1 input value bit assignments are:



Bits [63:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAAE1 instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VAAE1	000	1000	0111	011

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VAAE1	x	x	0	-	WO	n/a	WO
VAAE1	0	0	1	-	WO	WO	WO
VAAE1	0	1	1	-	n/a	WO	WO
VAAE1	1	0	1	-	WO	WO	WO
VAAE1	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.12 TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS characteristics are:

Purpose

Invalidate stage 1 TLB entries for the given VA and, if applicable, the current VMID on all PEs in the same Inner Shareable domain, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VAA](#).

Configurations

There are no configuration notes.

Attributes

TLBI VAAE1IS is a 64-bit system operation.

Field descriptions

The TLBI VAAE1IS input value bit assignments are:



Bits [63:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAAE1IS instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VAAE1IS	000	1000	0011	011

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VAAE1IS	x	x	0	-	WO	n/a	WO
VAAE1IS	0	0	1	-	WO	WO	WO
VAAE1IS	0	1	1	-	n/a	WO	WO
VAAE1IS	1	0	1	-	WO	WO	WO
VAAE1IS	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.13 TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1 characteristics are:

Purpose

Invalidate stage 1 TLB entries for the final level of translation table walk for the given VA and, if applicable, the current VMID, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VAAL](#).

Configurations

There are no configuration notes.

Attributes

TLBI VAALE1 is a 64-bit system operation.

Field descriptions

The TLBI VAALE1 input value bit assignments are:



Bits [63:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAALE1 instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VAALE1	000	1000	0111	111

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VAALE1	x	x	0	-	WO	n/a	WO
VAALE1	0	0	1	-	WO	WO	WO
VAALE1	0	1	1	-	n/a	WO	WO
VAALE1	1	0	1	-	WO	WO	WO
VAALE1	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.14 TLBI VAALE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAALE1IS characteristics are:

Purpose

Invalidate stage 1 TLB entries for the final level of translation table walk for the given VA and, if applicable, the current VMID, on all PEs in the same Inner Shareable domain, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VAAL](#).

Configurations

There are no configuration notes.

Attributes

TLBI VAALE1IS is a 64-bit system operation.

Field descriptions

The TLBI VAALE1IS input value bit assignments are:



Bits [63:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAALE1IS instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VAALE1IS	000	1000	0011	111

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VAALE1IS	x	x	0	-	WO	n/a	WO
VAALE1IS	0	0	1	-	WO	WO	WO
VAALE1IS	0	1	1	-	n/a	WO	WO
VAALE1IS	1	0	1	-	WO	WO	WO
VAALE1IS	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.15 TLBI VAE1, TLB Invalidate by VA, EL1

The TLBI VAE1 characteristics are:

Purpose

Invalidate stage 1 TLB entries for the given VA and, as applicable, the specified ASID and the current VMID, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VA](#).

Configurations

There are no configuration notes.

Attributes

TLBI VAE1 is a 64-bit system operation.

Field descriptions

The TLBI VAE1 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. When invalidating entries for a translation regime for which an ASID is valid:

- Any TLB entries that match the ASID value and VA value will be affected by this operation.
- Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

Bits [47:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE1 instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VAE1	000	1000	0111	001

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VAE1	x	x	0	-	WO	n/a	WO
VAE1	0	0	1	-	WO	WO	WO
VAE1	0	1	1	-	n/a	WO	WO
VAE1	1	0	1	-	WO	WO	WO
VAE1	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.16 TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable

The TLBI VAE1IS characteristics are:

Purpose

Invalidate stage 1 TLB entries for the given VA and, as applicable, the specified ASID and the current VMID, on all PEs in the same Inner Shareable domain, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VA](#).

Configurations

There are no configuration notes.

Attributes

TLBI VAE1IS is a 64-bit system operation.

Field descriptions

The TLBI VAE1IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. When invalidating entries for a translation regime for which an ASID is valid:

- Any TLB entries that match the ASID value and VA value will be affected by this operation.
- Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

Bits [47:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VAE1IS instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VAE1IS	000	1000	0011	001

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VAE1IS	x	x	0	-	WO	n/a	WO
VAE1IS	0	0	1	-	WO	WO	WO
VAE1IS	0	1	1	-	n/a	WO	WO
VAE1IS	1	0	1	-	WO	WO	WO
VAE1IS	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.17 TLBI VALE1, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1 characteristics are:

Purpose

Invalidate stage 1 TLB entries for the final level of translation table walk for the given VA and, as applicable, the specified ASID and the current VMID, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VAL](#).

Configurations

There are no configuration notes.

Attributes

TLBI VALE1 is a 64-bit system operation.

Field descriptions

The TLBI VALE1 input value bit assignments are:



ASID, bits [63:48]

ASID value to match. When invalidating entries for a translation regime for which an ASID is valid:

- Any TLB entries that match the ASID value and VA value will be affected by this operation.
- Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

Bits [47:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE1 instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VALE1	000	1000	0111	101

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VALE1	x	x	0	-	WO	n/a	WO
VALE1	0	0	1	-	WO	WO	WO
VALE1	0	1	1	-	n/a	WO	WO
VALE1	1	0	1	-	WO	WO	WO
VALE1	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.18 TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS characteristics are:

Purpose

Invalidate stage 1 TLB entries for the final level of translation table walk for the given VA and, as applicable, the specified ASID and the current VMID, on all PEs in the same Inner Shareable domain, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VAL](#).

Configurations

There are no configuration notes.

Attributes

TLBI VALE1IS is a 64-bit system operation.

Field descriptions

The TLBI VALE1IS input value bit assignments are:



ASID, bits [63:48]

ASID value to match. When invalidating entries for a translation regime for which an ASID is valid:

- Any TLB entries that match the ASID value and VA value will be affected by this operation.
- Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

Bits [47:44]

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing the TLBI VALE1IS instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VALE1IS	000	1000	0011	101

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VALE1IS	x	x	0	-	WO	n/a	WO
VALE1IS	0	0	1	-	WO	WO	WO
VALE1IS	0	1	1	-	n/a	WO	WO
VALE1IS	1	0	1	-	WO	WO	WO
VALE1IS	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TTLB](#)==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

B12.6.19 TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1 characteristics are:

Purpose

Invalidate all stage 1 TLB entries for the current VMID, if applicable, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VMALL](#).

Configurations

There are no configuration notes.

Attributes

TLBI VMALLE1 is a 64-bit system operation.

Field descriptions

The TLBI VMALLE1 instruction ignores the value in the register specified by the instruction used to execute this instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLE1 instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VMALLE1	000	1000	0111	000

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VMALLE1	x	x	0	-	WO	n/a	WO
VMALLE1	0	0	1	-	WO	WO	WO
VMALLE1	0	1	1	-	n/a	WO	WO
VMALLE1	1	0	1	-	WO	WO	WO
VMALLE1	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#). Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TTLB==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TTLB==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

RETIRED

B12.6.20 TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS characteristics are:

Purpose

Invalidate all stage 1 TLB entries for the current VMID, if applicable, on all PEs in the same Inner Shareable domain, in the following translation regime:

- In Secure state, and in Non-secure state, when [HCR_EL2](#).{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
- When [HCR_EL2](#).{E2H, TGE} is {1, 1}, the EL2&0 translation regime.

If EL3 is implemented, the value of [SCR_EL3](#).NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see [VMALL](#).

Configurations

There are no configuration notes.

Attributes

TLBI VMALLE1IS is a 64-bit system operation.

Field descriptions

The TLBI VMALLE1IS instruction ignores the value in the register specified by the instruction used to execute this instruction. Software does not have to write a value to the register before issuing this instruction.

Executing the TLBI VMALLE1IS instruction

This instruction can be executed using TLBI with the following syntax:

TLBI <tlbi_op>, <Xt>

This syntax is encoded with the following settings in the instruction encoding:

<tlbi_op>	op1	CRn	CRm	op2
VMALLE1IS	000	1000	0011	000

Accessibility

The instruction is executable as follows:

<tlbi_op>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
VMALLE1IS	x	x	0	-	WO	n/a	WO
VMALLE1IS	0	0	1	-	WO	WO	WO
VMALLE1IS	0	1	1	-	n/a	WO	WO
VMALLE1IS	1	0	1	-	WO	WO	WO
VMALLE1IS	1	1	1	-	n/a	WO	WO

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HCR_EL2.TTLB==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HCR_EL2.TTLB==1, Non-secure execution of this instruction at EL1 is trapped to EL2.

RETIRED

B12.7 ARMv8.0 sections relating to these registers

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* are included in this supplement to complement the register descriptions:

- [Mismatched memory attributes](#).
- [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#).
- [Asynchronous exception routing on page B12-550](#).
- [Address tagging in AArch64 state on page B12-551](#).
- [Scope of the A64 TLB maintenance instructions on page B12-553](#).
- [Invalidation of TLB entries from stage 2 translations on page B12-556](#).
- [Events, event numbers, and mnemonics on page B12-557](#).
- [Operation of the CompareValue views of the timers on page B12-557](#).
- [Operation of the TimerValue views of the timers on page B12-558](#).
- [Reserved values in System and memory-mapped registers and translation table entries on page B12-558](#).

B12.7.1 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see Chapter D4 The AArch64 Virtual Memory System Architecture in the ARMARM.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type, Device or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

———— **Note** ————

The terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory location is accessed with mismatched attributes the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory location might be lost. This means:
 - A read of the memory location by one agent might not return the value most recently written to that memory location by the same agent.
 - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Re-ordering.

For the following situations, when a physical memory location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. If the only memory type mismatch associated with a memory location across all users of the memory location is between different types of Device memory, then all accesses might take the properties of the weakest Device memory type.
2. Any agent that reads that memory location using the same common definition of the shareability and cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all of the following conditions are met:
 - All aliases to the memory location with write permission both use a common definition of the shareability and cacheability attributes for the memory location, and either:
 - Have the inner cacheability attribute the same as the outer cacheability attribute.
 - In the Non-secure EL1&0 translation regime, have [HCR_EL2.MI0CNCE](#) set to 0.
 - All aliases to a memory location use a definition of the shareability attributes that encompasses all the agents with permission to access the location.
3. The possible software-visible effects caused by mismatched attributes for a memory location are defined more precisely if all of the mismatched attributes define the memory location as one of:
 - Any Device memory type.
 - Normal Inner Non-cacheable, Outer Non-cacheable memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:

- Possible loss of properties derived from the memory type when multiple agents attempt to access the memory location.
- Possible reordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to the location, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable locations between agents in different shareability domains. This means:
 - Before writing to a location not using the Write-Back attribute, software must invalidate, or clean, a location from the caches if any agent might have written to the location with the Write-Back attribute. This avoids the possibility of overwriting the location with stale data.
 - After writing to a location with the Write-Back attribute, software must clean the location from the caches, to make the write visible to external memory.
 - Before reading the location with a cacheable attribute, software must invalidate the location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
 - Executing a DMB barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same memory location that use different attributes.

In all cases:

- Location refers to any byte within the current coherency granule.
- A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
- In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address.

Note

With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a location mean that multiple cacheable accesses to the location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
 - Each PE that accesses the location with a cacheable attribute performs a clean and invalidate of the location before and after accessing that location.
 - A DMB barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory location that use different attributes.

Note

The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a location, and the accesses from the different agents have different memory attributes associated with the location, the exclusive monitor state becomes UNKNOWN.

ARM strongly recommends that software does not use mismatched attributes for aliases of the same location. An implementation might not optimize the performance of a system that uses mismatched aliases.

B12.7.2 Synchronous exception prioritization for exceptions taken to AArch64

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. For exceptions taken to an Exception level that is using AArch64, these are prioritized as follows, where 1 is the highest priority.

Note

The priority numbering in this list only shows the relative priorities of exceptions taken to an Exception level that is using AArch64. This numbering has no global significance and, for example, does not correlate with the equivalent AArch32 list in [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#).

1. Software Step exceptions.
2. Misaligned PC exceptions.
3. Instruction Abort exceptions.
4. Breakpoint exceptions or Address Matching Vector Catch exceptions.

Vector Catch exceptions are only taken from AArch32 state.

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#). This means that it is outside the scope of the description of this section.

5. Illegal Execution state exceptions.

6. Exceptions taken from EL1 to EL2 because of one of the following configuration settings:
- HSTR_EL2.Tn.
 - HCR_EL2.TIDCP.

Note

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are HSTR.Tn and HCR.TIDCP.

7. Undefined Instruction exceptions that occur as a result of one or more of the following:
- An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
 - An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
 - Debug state execution of an instruction encoding that is unallocated in Debug state.
 - Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.
 - Execution of an HVC instruction, when HVC instructions are disabled by SCR_EL3.HCE or HCR_EL2.HCD.
 - Execution of an MSR or MRS instruction to SP_EL0 when the value of SPSel is 0.
 - Execution of an HLT instruction when HLT instructions are disabled by EDSCR.HDE.
 - In Debug state:
 - Execution of a DCP51 instruction in Non-secure EL0 when HCR_EL2.TGE is 1.
 - Execution of a DCP52 instruction in EL1 or EL0 when SCR_EL3.NS is 0 or when EL2 is not implemented.
 - Execution of a DCP53 instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
 - When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.
 - When executing in AArch32 state, execution of an instruction that is UNDEFINED as a result of any of:
 - Being in an IT block when SCTLR_EL1.ITD is 1.
 - Executing a SETEND instruction executed SCTLR_EL1.SED.
 - Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR_EL1.CP15BEN is 0.

Note

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are SCTLR.{ITD, SED, CP15BEN}, with additional controls HSCTLR.{ITD, SED, CP15BEN}.

- When executing in AArch32 state, execution of an instruction that is UNDEFINED because at least one of FPCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported.

Note

- This case applies only when EL0 is using AArch32 and EL1 is using AArch64. The exception generated by the attempted execution at EL0 of the UNDEFINED instruction is taken to EL1 using AArch64.
- When EL1 is using AArch32, the corresponding controls are FPSCR.{Stride, Len}, and any exception generated by the attempted execution at EL0 or EL1 of an instruction that is UNDEFINED because of a nonzero {Stride, Len} value is taken to EL1 using AArch32.

8. Exceptions taken to EL1, or taken to EL2 because the value of HCR_EL2.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 1-7.

———— **Note** ————

When EL2 is using AArch32, the equivalent control for routing exceptions to EL2 is HCR.TGE.

9. Exceptions taken from EL0 to EL2 because of one of the following configuration settings:

- HSTR_EL2.Tn.
- HCR_EL2.TIDCP.

———— **Note** ————

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are HSTR.Tn and HCR.TIDCP.

10. Exceptions taken to EL2 because of configuration settings in the CPTR_EL2.

———— **Note** ————

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state, the equivalent controls are in the HCPTR.

11. Exceptions taken to EL2 because of one of the following configuration settings:

- Any setting in HCR_EL2, other than the TIDCP bit.
- Any setting in CNTHCTL_EL2.
- Any setting in MDCR_EL2.

———— **Note** ————

These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state, the equivalent controls are:

- Any setting in HCR, other than the TIDCP bit.
- Any setting in CNTHCTL or HDCR.

12. Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 1-11.

13. Exceptions caused by the SMC instruction being UNDEFINED because the value of SCR_EL3.SMD is 1.

14. Exceptions caused by the execution of an Exception generating instruction:

- For exceptions taken from AArch64 state, the section *Branches, Exception generating, and System instructions* in Chapter C3 of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* defines these instructions.
- When executing in AArch32 state, the exception-generating instructions are SVC, HVC, SMC, and BKPT.

15. Exceptions taken to EL3 because of configuration settings in the CPTR_EL3.

16. Exceptions taken to EL3 from Secure EL1 using AArch32, because of execution of the instructions listed in the section *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* in Chapter D1 of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

17. Exceptions taken to EL3 from EL0, EL1, or EL2 because of configuration settings in the MDCR_EL3.

18. Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 1-17.

19. Trapped floating-point exceptions, if supported.

20. Stack Pointer Alignment faults.

21. Data Abort exceptions other than a Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk. That is, any Data Abort exception that is not covered by item 23. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 23.
22. Watchpoint exceptions.
23. Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 21.

For items 21-23, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

———— **Note** ————

Exceptions generated by a translation table walk are reported and prioritized as either an Instruction Abort exception, priority 3 in this list, or a Data Abort exception, priority 21 in this list.

B12.7.3 Asynchronous exception routing

The following tables show the routing of physical interrupts when the highest implemented Exception level is using AArch64.

In the tables, C indicates that the interrupt is not taken, regardless of the Process state interrupt mask.

Table B12-1 Routing when both EL3 and EL2 are implemented

SCR_EL3.EA SCR_EL3.IRQ SCR_EL3.FIQ	SCR_EL3.RW	AMO ^a IMO ^a FMO ^a	Target Exception level when executing in:					
			Non-secure			Secure		
			EL0	EL1	EL2	EL0	EL1	EL3
0	0	0	EL1	EL1	EL2	EL1	EL1	C
	X	1	EL2	EL2	EL2	EL1	EL1	C
	1	0	EL1	EL1	C	EL1	EL1	C
1	X	X	EL3	EL3	EL3	EL3	EL3	EL3

a. If EL2 is using AArch64, these are the HCR_EL2.{AMO, IMO, FMO} control bits. If EL2 is using AArch32, these are the HCR{AMO, IMO, FMO} control bits. If HCR_EL2.TGE or HCR.TGE is 1, these bits are treated as being 1 other than for a direct read.

Table B12-2 Routing when EL3 is implemented and EL2 is not implemented

SCR_EL3.EA SCR_EL3.IRQ SCR_EL3.FIQ	Target Exception level when executing in:				
	Non-secure		Secure		
	EL0	EL1	EL0	EL1	EL3
0	EL1	EL1	EL1	EL1	C
1	EL3	EL3	EL3	EL3	EL3

Table B12-3 Routing when EL3 is not implemented and EL2 is implemented

HCR_EL2.AMO ^a HCR_EL2.IMO ^a HCR_EL2.FMO ^a	Target Exception level when executing in:		
	Non-secure		
	EL0	EL1	EL2
1	EL2	EL2	EL2
0	EL1	EL1	C

a. If HCR_EL2.TGE is 1, these bits are treated as being 1 other than for a direct read.

B12.7.4 Address tagging in AArch64 state

In AArch64 state, the ARMv8 architecture supports tagged addresses for data values. In these cases the top eight bits of the virtual address are ignored when determining:

- Whether the address causes a Translation fault from being out of range if the translation system is enabled.
- Whether the address causes an Address size fault from being out of range if the translation system is not enabled.
- Whether the address requires invalidation when performing a TLB invalidation instruction by address.

The use of address tags is controlled as follows:

For addresses using the VMSAv8-64 EL1&0 or EL2 translation regime when HCR_EL2.E2H == 1

The value of bit[55] of the VA determines the register bit that controls the use of address tags, as follows:

VA[55]==0

TCR_ELx.TBI0 determines whether address tags are used. If stage 1 translation is enabled, TTBR0_ELx holds the base address of the translation tables used to translate the address.

VA[55]==1

TCR_ELx.TBI1 determines whether address tags are used. If stage 1 translation is enabled, TTBR1_ELx holds the base address of the translation tables used to translate the address.

For addresses using the VMSAv8-64 EL2 translation regime when HCR_EL2.E2H == 0

TCR_EL2.TBI determines whether address tags are used. If stage 1 translation is enabled, TTBR0_EL2 holds the base address of the translation tables used to translate the address.

For addresses using the VMSAv8-64 EL3 translation regime

TCR_EL3.TBI determines whether address tags are used. If stage 1 translation is enabled, TTBR0_EL3 holds the base address of the translation tables used to translate the address.

———— Note ————

The TCR_ELx.TBI n bits determine whether address tags are used regardless of whether the corresponding translation regime is enabled.

An address tag enable bit also has an effect on the PC value in the following cases:

- Any branch or procedure return within the controlled Exception level.
- On taking an exception to the controlled Exception level, regardless of whether this is also the Exception level from which the exception was taken.
- On performing an exception return to the controlled Exception level, regardless of whether this is also the Exception level from which the exception return was performed.

- Exiting from debug state to the controlled Exception level.

———— **Note** ————

As an example of what is meant by the *controlled Exception level*, TCR_EL2.TBI controls this effect for:

- A branch or procedure return within EL2.
- Taking an exception to EL2.
- Performing an exception return or a debug state exit to EL2.

The effect of the controlling TBI{n} bit is:

For EL0 or EL1 or EL2, when HCR_EL2.E2H == 1

If the controlling TBI n bit for the address being loaded into the PC is set to 1, then bits[63:56] of the PC are forced to be a sign-extension of bit[55] of that address.

For EL2 or EL3, when HCR_EL2.E2H == 0

If the controlling TBI bit for the address being loaded into the PC is set to 1, then bits[63:56] of the PC are forced to be 0x00.

The AddrTop() pseudocode function shows the algorithm determining the most significant bit of the VA, and therefore whether the virtual address is using tagging.

———— **Note** ————

The required behavior prevents a tagged address being propagated to the program counter.

When address tagging is enabled for an address that causes a Data Abort or a Watchpoint, the address tag is included in the virtual address returned in the FAR.

Relaxation of the tagged address handling requirements on an Illegal exception return

The AddrTop() pseudocode function does not cover a relaxation to the requirements for tagged address handling that applies to an Illegal exception return. In the case of an Illegal exception return, it is IMPLEMENTATION DEFINED whether the exception return targets:

- The Exception level indicated by the current SPSR at the time of the exception return.
- The Exception level at which the exception return instruction was executed.

The AArch64.ExceptionReturn() pseudocode function includes this IMPLEMENTATION DEFINED choice.

———— **Note** ————

- The TCR_ELx.TBI x fields have the effect shown in the AArch64.ExceptionReturn() pseudocode regardless of whether the corresponding translation regime is enabled.
 - In the case of an Illegal exception return, the tag bits of the address can be propagated to the PC if all of the following apply:
 - The implementation treats the target_exception_level as being the Exception level that was described in the SPSR at the time of the exception return.
 - For the Exception level that was described in the SPSR at the time of the exception return, the value of TCR_ELx.TBI is 0.
 - In the Exception level that the exception was taken from, the value of TCR_ELx.TBI is 1.
- In all other cases, the tag bits cannot be propagated to the PC.

B12.7.5 Scope of the A64 TLB maintenance instructions

The TLB invalidation instruction <type> affects the different possible cached entries in the TLB as follows:

ALL The invalidation applies to all cached copies of the stage 1 and stage 2 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with the state specified by SCR_EL3.NS.

For entries from the Non-secure EL1&0, ALL applies to entries with any VMID.

The invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

Note

This means that, for a translation regime for which an ASID is valid, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

VMALL The invalidation applies to all cached copies of the stage 1 translation table entries, from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- The current VMID, for the Non-secure EL1&0 translation regime.

For all entries that meet these conditions, the invalidation applies to:

- Entries above the final level of lookup.
- Entries at the final level of lookup.

Note

This means that, for a translation regime for which an ASID is valid, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

VMALL is valid for:

- EL1.
- EL2&0, when HCR_EL2.{E2H, TGE} is {1, 1}.

VMALLS12 The invalidation applies to all cached copies of the stage 1 and stage 2 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- The current VMID, for the Non-secure EL1&0 translation regime.

For all entries that meet these conditions, the invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

Note

This means that, for a translation regime for which an ASID is valid, the invalidation applies to both:

- Global entries.
- Non-global entries with any ASID.

VMALLS12 is valid for:

- EL1.
- EL2&0, when HCR_EL2.{E2H, TGE} is {1, 1}.

If EL2 is not implemented, or if the TLBI VMALLS12 instruction is executed when the value of SCR_EL3.NS is 0, the instruction is not UNDEFINED but it has the same effect as TLBI VMALL. This is because there are no stage 2 translations to invalidate.

ASID The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- The current VMID, for the Non-secure EL1&0 translation regime.

For an entry from the EL1&0 or EL2&0 translation regime that meets these conditions, the invalidation applies only if either:

- The entry is from a level of lookup above the final level and matches the specified ASID.
- The entry is a non-global entry from the final level of lookup and matches the specified ASID.

ASID is valid for:

- EL1.
- EL2&0, when HCR_EL2.{E2H, TGE} is {1, 1}.

VA The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- The current VMID, for the Non-secure EL1&0 translation regime.

For an entry from a translation regime for which an ASID is valid that meets these conditions, the invalidation applies if one of the following applies:

- The entry is from a level of lookup above the final level and matches the specified ASID.
- The entry is a global entry from the final level of lookup.
- The entry is a non-global entry from the final level of lookup that matches the specified ASID.

VAL The invalidation applies to all cached copies of the stage 1 translation table entry from the final level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- The current VMID, for the Non-secure EL1&0 translation regime.

For an entry from a translation regime for which an ASID is valid that meets these conditions, the invalidation applies only if either:

- The entry is a global entry from the final level of lookup.
- The entry is a non-global entry from the final level of lookup that matches the specified ASID.

VAA	<p>The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:</p> <ul style="list-style-type: none"> • The Security state specified by SCR_EL3.NS. • The current VMID, for the Non-secure EL1&0 translation regime. <p>For entries that meet these conditions, the invalidation applies to all of:</p> <ul style="list-style-type: none"> • All entries above the final level of lookup. • All entries at the final level of lookup. <p style="text-align: center;">———— Note ————</p> <p>This means that, for a translation regime for which an ASID is valid, the invalidation applies to both:</p> <ul style="list-style-type: none"> — Global entries. — Non-global entries with any ASID.
VAAL	<p>The invalidation applies to all cached copies of the stage 1 translation table entry from the final level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:</p> <ul style="list-style-type: none"> • The Security state specified by SCR_EL3.NS. • The current VMID, for the Non-secure EL1&0 translation regime. <p>For entries that meet these conditions, the invalidation applies to all entries at the final level of lookup.</p> <p style="text-align: center;">———— Note ————</p> <p>This means that, for a translation regime for which an ASID is valid, the invalidation applies to both:</p> <ul style="list-style-type: none"> • Global entries. • Non-global entries with any ASID.
IPAS2	<p>The invalidation applies to all cached copies of the stage 2 translation table entries from any level of the translation table walk required to translate the specified IPA, that both:</p> <ul style="list-style-type: none"> • Are held in TLB caching structures holding stage 2 only entries. • Would be used with the current VMID. <p>It is not required that this instruction invalidates TLB caching structures holding entries that combine stage 1 and stage 2 of the translation.</p> <p>The only translation regime to which this instruction can apply is the Non-secure EL1&0 translation regime.</p> <p>When executed with the SCR_EL3.NS=0, or in an implementation that does not implement EL2, this instruction is a NOP.</p> <p>For more information about the architectural requirements for the IPAS2 instruction see Invalidation of TLB entries from stage 2 translations on page B12-556.</p>
IPAS2L	<p>The invalidation applies to cached copies of the stage 2 translation table entry from the final level of the stage 2 translation table walk required to translate the specified IPA, that both:</p> <ul style="list-style-type: none"> • Are held in TLB caching structures holding stage 2 only entries. • Would be used with the current VMID. <p>It is not required that this instruction invalidates TLB caching structures holding entries that combine stage 1 and stage 2 of the translation.</p> <p>The only translation regime to which this instruction can apply is the Non-secure EL1&0 translation regime.</p>

When executed with the SCR_EL3.NS==0, or in an implementation that does not implement EL2, this instruction is a NOP.

For more information about the architectural requirements for the IPAS2L instruction see [Invalidation of TLB entries from stage 2 translations](#).

The entries that the invalidations apply to are not affected by the state of any other control bits involved in the translation process. Therefore, the following is a non-exhaustive list of control bits that do not affect how a TLB maintenance instruction updates the TLB entries:

In AArch64 SCTLR_EL1.M, SCTLR_EL2.M, SCTLR_EL3.{M, RW}, HCR_EL2.{VM, RW}, TCR_EL1.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}, TCR_EL2.{TG0, T0SZ}, TCR_EL3.{TG0, T0SZ}, VTCR_EL2.{SL0, T0SZ}, TTBR0_EL1.ASID, TTBR1_EL1.ASID.

In AArch32 SCTLR.M, HCR.VM, TTBCR.{EAE, PD1, PD0, N, EPD1, T1SZ, EPD0, T0SZ, A1}, HTCR.T0SZ, VTCR.{SL0, T0SZ}, TTBR0.ASID, TTBR1.ASID, CONTEXTIDR.ASID.

————— Note —————

- ARM expects most TLB maintenance performed by an operating system to occur to the last level entries of the stage 1 translation table walks, and the purpose of the address-based TLB invalidation instructions where the invalidation need only apply to caching of entries returned from the last level of translation table walk of stage 1 translation is to avoid unnecessary loss of the intermediate caching of the translation table entries. Similarly, for stage 2 translations ARM expects that most TLB maintenance performed by a hypervisor for a given Guest operation system will affect only the last level entries of the stage 2 translations. Therefore, similar capability is provided for instructions that invalidate single stage 2 entries.
- The architecture permits the invalidation of entries in TLB caching structures at any time, so for each of these instructions the definition specifies only the minimum set of entries that must be invalidated from TLB caching structures, and an implementation might choose to invalidate more entries. In general, for best performance, ARM recommends not invalidating entries that are not required to be invalidated.
- Dependencies on the VMID for the Non-secure EL1&0 translation regime apply even when HCR_EL2.VM is set to 0. Because the architecture does not require the VTTBR_EL2.VMID field to be reset in hardware, the reset routine of each active PE must initialize VTTBR_EL2 to a common value such as 0, even if stage 2 translation is not in use.

B12.7.6 Invalidation of TLB entries from stage 2 translations

The architectural requirements of the IPAS2 instruction are that:

1. The following code is sufficient to invalidate all cached copies of the stage 2 translation of the IPA held in Xt for the current VMID, with the corresponding requirement for the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VMALLE1
```
2. The following code is sufficient to invalidate all cached copies of the stage 2 translations of the IPA held in Xt used to translate the virtual address VA (and the specified ASID when executing TLBI VAE1) held in Xt2, with the corresponding requirement for the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VAE1, Xt2 ; or TLBI VAAE1, Xt2
```
3. The following code is sufficient to invalidate all cached copies of the stage 2 translations of the IPA held in Xt used to translate the IPA produced by the last level of stage 1 translation table lookup for the virtual address VA (and ASID when executing TLBI VALE1) held in Xt2, with the corresponding requirement for the broadcast versions of the instructions:

```
TLBI IPAS2E1, Xt
DSB
TLBI VALE1, Xt2 ; or TLBI VAALE1, Xt2
```

———— **Note** ————

Sequences 1, 2, and 3 must use the TLBI IPAS2E1 instruction even when Non-secure EL1&0 stage 1 translation is disabled.

Equivalent architectural requirements apply to the IPAS2L instruction, except that the only TLB entries that must be invalidated by an IPAS2L instruction are those that come from the final level of the translation table lookup.

B12.7.7 Events, event numbers, and mnemonics

This is a section from Chapter D5 The Performance Monitors Extension from the ARMv8 ARM. The contents of this section are unchanged, except for:

- The event number space is extended from 12 bits in ARMv8.0 to 16 bits in ARMv8.1. See [Extended event number space on page B9-74](#) for more information.
- In an ARMv8.1 implementation, in addition to the events required by PMMUv3, the STALL_FRONTEND and STALL_BACKEND events must be implemented. For more information, see [Required events on page B9-74](#).

B12.7.8 Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into a CompareValue register. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTPS_CTL_EL1, or CNTV_CTL_EL0. For CNTP_CTL_EL0, the asserted interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch32 register CNTP_CTL.

The operation of this view of a timer is:

$$\text{TimerConditionMet} = (((\text{Counter}[63:0] - \text{Offset}[63:0])[63:0] - \text{CompareValue}[63:0]) \geq 0)$$

Where:

TimerConditionMet Is TRUE if the timer condition for this counter is met, and FALSE otherwise.

Counter The physical counter value, that can be read from the CNTPCT_EL0 register.

———— **Note** ————

The virtual counter value, that can be read from the CNTVCT_EL0 register, is the value:
(Counter - Offset)

Offset For a physical timer it is zero, and for the virtual timer it is the virtual offset, held in the CNTVOFF_EL2 register.

CompareValue The value of the appropriate CompareValue register, CNTP_CVAL_EL0, CNTHP_CVAL_EL2, CNTPS_CVAL_EL1, or CNTV_CVAL_EL0.

In this view of a timer, Counter, Offset, and CompareValue are all 64-bit unsigned values.

———— **Note** ————

This means that a timer with a CompareValue of, or close to, 0xFFFF_FFFF_FFFF_FFFF might never trigger. However, there is no practical requirement to use values close to the counter wrap value.

B12.7.9 Operation of the TimerValue views of the timers

The TimerValue view of a timer operates as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHV_CTL_EL2, CNTPS_CTL_EL1, or CNTV_CTL_EL0.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads $\text{TimerValue} = (\text{CompareValue} - (\text{Counter} - \text{Offset})) [31:0]$

Writes $\text{CompareValue} = ((\text{Counter} - \text{Offset}) [63:0] + \text{SignExtend}(\text{TimerValue})) [63:0]$

Where the arguments have the definitions used in [Operation of the CompareValue views of the timers on page B12-557](#), and in addition:

TimerValue The value of a TimerValue register, CNTP_TVAL_EL0, CNTHP_TVAL_EL2, CNTHV_TVAL_EL2, CNTPS_TVAL_EL1, or CNTV_TVAL_EL0.

In this view of a timer, all values are signed, in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

———— **Note** ————

- [Operation of the CompareValue views of the timers on page B12-557](#) gives a strict definition of TimerConditionMet. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from 0x80000000, the TimerValue view can be used as giving an effect equivalent to:

$$\text{TimerConditionMet} = (\text{TimerValue} \cdot 0)$$

- Programming TimerValue to a negative number with magnitude greater than (Counter–Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays meeting the timer condition for an extremely long period of time.

B12.7.10 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The unallocated value maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.

———— **Note** ————

These constraints are identical to those for the equivalent AArch32 definitions, as given in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Part C

ARMv8.1 Changes in the AArch32 Architecture

RETIRED

Chapter C1

A32/T32 Advanced SIMD Instructions for Rounding Double Multiply Add/Subtract

This chapter describes the Rounding double multiply add and Rounding double multiply subtract instructions added to the T32 and A32 Advanced SIMD instruction sets. It contains the following section:

- [About the new instructions on page C1-562.](#)

C1.1 About the new instructions

The following instructions are added to the Advanced SIMD instruction set:

- VQRDMLAH, Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half.
- VQRDMLSH, Vector Saturating Rounding Doubling Multiply Subtract Returning High Half.

C1.1.1 Behavior in Debug state

In Debug state, these instructions are CONSTRAINED UNPREDICTABLE, and the behavior is one of the following:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction executes as in Non-debug state.

C1.1.2 Identification mechanism

The [ID_ISAR5_EL1.RDM](#) and [ID_ISAR5.RDM](#) fields identify the support for the new Advanced SIMD instructions.

C1.1.3 See also

In this supplement

- [Advanced SIMD data-processing on page C4-572.](#)
- Instructions:
 - [VQRDMLAH.](#)
 - [VQRDMLSH.](#)
- Registers:
 - [ID_ISAR5.RDM.](#)

In the ARM Architecture Reference Manual

The following chapters will be updated:

The T32 Instruction Set Encoding

- Advanced SIMD data-processing.
- Advanced SIMD three registers of the same length.
- Advanced SIMD two registers and a scalar.

The T32 Instruction Set Encoding

- Advanced SIMD data-processing.
- Advanced SIMD three registers of the same length.
- Advanced SIMD two registers and a scalar.

Chapter C2

AArch32 Privileged Access Never

This chapter describes the addition of a Privileged access never field to PSTATE. It contains the following section:

- [About the Privileged Access Never bit on page C2-564.](#)

C2.1 About the Privileged Access Never bit

A new PAN (Privileged Access Never) state bit is added to PSTATE. When the value of this bit is 1, any access from PL1 or higher to a memory address that is accessible at PL0 generates a Permission fault. A corresponding PAN bit is added to [CPSR](#), [DSPSR](#), and [SPSR](#).

The PAN bit can be written in the [CPSR](#) using an MSR instruction at PL1 or higher. Data writes to the PAN bit in [CPSR](#) using an MSR instruction at PL0 are ignored. The value that is returned for an MRS instruction of [CPSR](#) from PL0 is UNKNOWN. In keeping with all other writes to the [CPSR](#), other than for instruction fetches, the effect of the PAN bit does not need to be explicitly synchronized.

When the value of the PAN bit is 0, the translation system is the same as in ARMv8.0.

The PAN bit has no effect on:

- Data Cache instructions.
- Address translation instructions.
- Unprivileged instructions, LDRBT, LDRHT, LDRT, LDRSBT, LDRSHT, STRBT, STRHT, STRT, STRSBT, and STRSHT.
- Instruction accesses.
- Manager domains.

If access is disabled, then the access will give rise to a stage 1 Permission fault.

On an exception taken from AArch32 to AArch32, PSTATE.PAN is copied to SPSR.PAN.

On an exception return from AArch32:

- PSTATE.PAN is copied to SPSR_ELx.PAN, when the target Exception level is in AArch64 state.
- [SPSR](#).PAN is copied to PSTATE.PAN, when the target Exception level is in AArch32 state.

On entry to Debug state, PSTATE.PAN is copied to [DSPSR](#).PAN.

On exit from Debug state, [DSPSR](#).PAN is copied to PSTATE.PAN.

C2.1.1 Behavior in Debug state

In Debug state, the behavior of instructions for accessing PSTATE.PAN bit is CONSTRAINED UNPREDICTABLE, and have one of the following behaviors:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction executes as in Non-debug state, setting DSPSR and DLR to UNKNOWN values.

C2.1.2 Identification mechanism

The ID_MMFR3.PAN field identifies the support for the Privileged Access Never bit.

C2.1.3 See also

In this supplement

- Instructions:
 - [SETPAN](#).
- Registers:
 - [CPSR](#).PAN.
 - [ID_MMFR3](#).PAN.
 - [SCTLR](#).SPAN.
 - [SPSR](#).PAN.
 - [SPSR_abt](#).PAN.

- [SPSR_fiq](#).PAN.
- [SPSR_hyp](#).PAN.
- [SPSR_irq](#).PAN.
- [SPSR_mon](#).PAN.
- [SPSR_svc](#).PAN.
- [SPSR_und](#).PAN.
- [DPSR](#).PAN.

In the ARM Architecture Reference Manual

- Process state, PSTATE.
- The Current Program Status Register, CPSR.
- Memory access control section in The AArch32 Virtual Memory System Architecture chapter.

RETIRED

RETIRED

Chapter C3

AArch32 Performance Monitors Extension

This chapter describes the changes to the Performance Monitors Extension introduced with ARMv8.1. It contains the following section:

- [Changes to the Performance Monitors Extension on page C3-568.](#)

C3.1 Changes to the Performance Monitors Extension

The OPTIONAL Performance Monitors Extension is enhanced to:

- Provide a new control to disable event counting at EL2. A control bit HPMD is added to [HDCR](#) to prohibit event counting at EL2.
- The event number space is extended to 16 bits to allow additional IMPLEMENTATION DEFINED event types and extend the reserved space for future additions to the architecturally-defined event types.
- In an ARMv8.1 implementation, in addition to the events required by PMUv3, the STALL_FRONTEND and STALL_BACKEND events must be implemented. For more information, see [Required events](#).

C3.1.1 Extended event number space

The event number space is extended to 16 bits, and is defined as:

0x0000-0x003F and 0x4000-0x403F

Common architectural and microarchitectural events, discoverable using PMCEID<n>.

0x0040-0x00BF and 0x4040-0x40BF

ARM recommended common architectural and microarchitectural events. These are IMPLEMENTATION DEFINED.

0x8000-0x80BF and 0xC000-0xC0BF

Reserved.

All other values

IMPLEMENTATION DEFINED events.

To address this extended number space, the [PMEVTYPER<n>](#).evtCount is extended to 16 bits.

C3.1.2 Required events

PMUv3 requires that an implementation includes the following common events:

- 0x0000, SW_INCR, Instruction architecturally executed, condition code check pass, software increment.
- 0x0003, L1D_CACHE_REFILL, Attributable Level 1 data cache refill.

Note

Event 0x0003 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0004, L1D_CACHE, Attributable Level 1 data cache access.

Note

Event 0x0004 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0010, BR_MIS_PRED, Mispredicted or not predicted branch speculatively executed.

Note

Event 0x0010 is only required if the implementation includes program-flow prediction. However, ARM recommends that the event is implemented as described in the section Common microarchitectural event numbers in Chapter D5 The Performance Monitors Extension of the ARM Architecture Reference Manual.

- 0x0011, CPU_CYCLES, Cycle.
- 0x0012, BR_PRED, Predictable branch speculatively executed.

———— **Note** ————

Event 0x0012 is only required if the implementation includes program-flow prediction. However, ARM recommends that the event is implemented as described in the section Common microarchitectural event numbers in Chapter D5 The Performance Monitors Extension of the ARMv8 ARM.

- At least one of:
 - 0x0008, INST_RETIRE, Instruction architecturally executed.
 - 0x001B, INST_SPEC, Operation speculatively executed.

———— **Note** ————

ARM strongly recommends that event 0x008 is implemented.

- 0x0023, STALL_FRONTEND, No operation issued due to the frontend. In ARMv8.1, this event must be implemented.
- 0x0024, STALL_BACKEND, No operation issued due to the backend. In ARMv8.1, this event must be implemented.

C3.1.3 Identification mechanism

The ID_DFR0.PerfMon and EDDFR.PMUVer fields describe the [PMEVTYPER<n>.evtCount](#) range.

C3.1.4 See also

In this supplement

- [HDCR.HPMD](#).
- [ID_DFR0.PerfMon](#).
- [PMCEID2](#).
- [PMCEID3](#).
- [PMCR](#).
- [PMEVTYPER<n>.evtCount](#).
- [PMCEID2](#) (External).
- [PMCEID3](#) (External).

In the ARM Architecture Reference Manual

- The Performance Monitors Extension chapter.
In the ARMv8 ARM, there is only one chapter on Performance Monitors Extension that covers both AArch64 and AArch32.
- Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events.

RETIRED

Chapter C4

A32/T32 Instruction Set Encoding

This chapter describes the encoding of the instructions that ARMv8.1 adds to the T32 and A32 instruction sets. It contains the following section:

- [Advanced SIMD data-processing on page C4-572.](#)

C4.1 Advanced SIMD data-processing

The [VQRDMLAH](#) and [VQRDMLSH](#) instructions are added to the Advanced SIMD data-processing instructions group under the Two registers and a scalar instruction class for the scalar form, and under the Three registers of the same length instruction class for the non-scalar form.

C4.1.1 Advanced SIMD three registers of the same length

This section describes the encoding of the Three registers of the same length instruction class. The encodings of the other instructions in this instruction class remain unchanged. Other encodings in this space are UNDEFINED.

The T32 encoding of these instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	U	1	1	1	1	0	D	size	Vn			Vd		opc		N	Q	M	o1		Vm

The A32 encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	U	0	D	size	Vn		Vd		opc		N	Q	M	o1		Vm	

The below table shows the allocation of encodings for the new instructions in this space.

Decode fields					Instruction page
opc	o1	U	size	Q	
1011	1	1	-	0	VQRDMLAH - 64-bit SIMD vector variant
1011	1	1	-	1	VQRDMLAH - 128-bit SIMD vector variant
1100	1	1	-	0	VQRDMLSH - 64-bit SIMD vector variant
1100	1	1	-	1	VQRDMLSH - 128-bit SIMD vector variant

C4.1.2 Advanced SIMD two registers and a scalar

This section describes the encoding of the Two registers and a scalar instruction class. The encodings of the other instructions in this instruction class remain unchanged. Other encodings in this space are UNDEFINED.

The T32 encoding of these instructions is:

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

The A32 encoding of these instructions is:

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	Q	1	D	size	Vn	Vd	opc	N	1	M	0	Vm					

If B = 0b11, see 'Advanced SIMD data-processing' for the T32 instruction set, or 'Advanced SIMD data-processing' for the A32 instruction set in the ARMv8 ARM.

Otherwise, the below table shows the allocation of encodings for the new instructions in this space.

Decode fields		Instruction page
opc	Q	
1110	0	VQRDMLAH - 64-bit SIMD vector variant
1111	0	VQRDMLSH - 64-bit SIMD vector variant
1110	1	VQRDMLAH - 128-bit SIMD vector variant
1111	1	VQRDMLSH - 128-bit SIMD vector variant

C4.1.3 Miscellaneous 16-bit instructions

The SETPAN instruction has been added to the Miscellaneous 16-bit instruction group.

See the Miscellaneous 16-bit instructions section in Chapter F3 The T32 Instruction Set Encoding of the ARMv8 ARM.

C4.1.4 See also

In the ARM Architecture Reference Manual

The following chapters of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* will be updated:

The T32 Instruction Set Encoding

- Advanced SIMD data-processing.
- Advanced SIMD three registers of the same length.
- Advanced SIMD two registers and a scalar.
- Miscellaneous 16-bit instructions.

The A32 Instruction Set Encoding

- Advanced SIMD data-processing.
- Advanced SIMD three registers of the same length.
- Advanced SIMD two registers and a scalar.
- Miscellaneous.

RETIRED

Chapter C5

A32 and T32 Instructions

This chapter describes the new T32 and A32 instructions introduced in ARMv8.1. It contains the following sections:

- [Alphabetical list of instructions on page C5-576.](#)
- [ARMv8.0 sections relating to these instructions on page C5-584.](#)

C5.1 Alphabetical list of instructions

This section lists the instructions added to the A32/T32 instruction set in ARMv8.1.

RETIRED

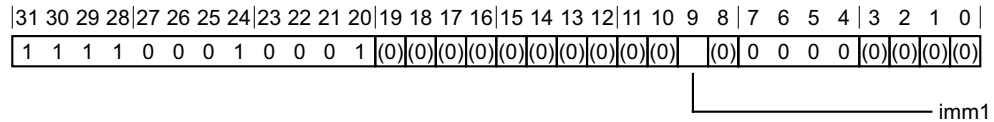
C5.1.1 SETPAN

Set Privileged Access Never writes a new value to PSTATE.PAN.

This instruction is available only in privileged mode and it is a NOP when executed in User mode.

A1

ARMv8.1



A1 variant

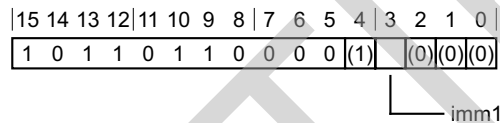
SETPAN{<q>} #<imm> // Cannot be conditional

Decode for this encoding

```
if !HavePANExt() then UNDEFINED;
value = imm1;
```

T1

ARMv8.1



T1 variant

SETPAN{<q>} #<imm> // Not permitted in IT block

Decode for this encoding

```
if !HavePANExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
value = imm1;
```

Assembler symbols

<q> See [Standard assembler syntax fields on page C5-585](#).

<imm> Is the unsigned immediate 0 or 1, encoded in the "imm1" field.

Operation for all encodings

```
EncodingSpecificOperations();
if PSTATE.EL != EL0 then
    PSTATE.PAN = value;
```

C5.1.2 VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see [Pseudocode description of saturation on page C5-584](#).

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and 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 on page C5-585](#).

A1

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```

if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

```

A2

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	0	1	Q	1	D	!=11	Vn	Vd	1	1	1	0	N	1	M	0	Vm				

size

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLAH{<q>}.<dt> {<Dd>}, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLAH{<q>}.<dt> {<Qd>}, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```

if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

```

T1

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```

if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

```

T2

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLAH{<q>}.<dt> {<Dd>}, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLAH{<q>}.<dt> {<Qd>}, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if !HaveQRDMLAExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);
```

Notes for all encodings

Related encodings: See "Advanced SIMD data-processing" for the T32 instruction set, or "Advanced SIMD data-processing" for the A32 instruction set.

Assembler symbols

<q>	See Standard assembler syntax fields on page C5-585 .
<dt>	Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values: S16 when size = 01 S32 when size = 10 The following encodings are reserved: <ul style="list-style-type: none"> size = 00. size = 11.
<Qd>	Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Dm[x]>	The scalar for either a quadword or a doubleword scalar operation. If <dt> is S16, Dm is restricted to D0-D7. If <dt> is S32, Dm is restricted to D0-D15.

Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = SInt(Elem[D[n+r],e,esize]);
        op3 = SInt(Elem[D[d+r],e,esize]) << esize;
        if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
        (result, sat) = SignedSatQ((op3 + 2*(op1*op2) + round_const) >> esize, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSR.QC = '1';
```

C5.1.3 VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see [Pseudocode description of saturation on page C5-584](#).

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and 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 on page C5-585](#).

A1

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```

if !HaveQRDMLAExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

```

A2

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> {<Dd>}, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLSH{<q>}.<dt> {<Qd>}, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```

if !HaveQRDMLAExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

```

T1

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```

if !HaveQRDMLAExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

```

T2

ARMv8.1

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

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> {<Dd>}, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLSH{<q>}.<dt> {<Qd>}, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```

if !HaveQRDMLAExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

```

Notes for all encodings

Related encodings: See "Advanced SIMD data-processing" for the T32 instruction set, or "Advanced SIMD data-processing" for the A32 instruction set.

Assembler symbols

<q>	See Standard assembler syntax fields on page C5-585 .
<dt>	Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values: S16 when size = 01 S32 when size = 10 The following encodings are reserved: <ul style="list-style-type: none"> size = 00. size = 11.
<Qd>	Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Dm[x]>	The scalar for either a quadword or a doubleword scalar operation. If <dt> is S16, Dm is restricted to D0-D7. If <dt> is S32, Dm is restricted to D0-D15.

Operation for all encodings

```

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = SInt(Elem[D[n+r],e,esize]);
        op3 = SInt(Elem[D[d+r],e,esize]) << esize;
        if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
        (result, sat) = SignedSatQ((op3 - 2*(op1*op2) + round_const) >> esize, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSR.QC = '1';

```

C5.2 ARMv8.0 sections relating to these instructions

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* are included in this supplement to complement the register descriptions.

C5.2.1 Pseudocode description of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N . This is supported in pseudocode by:

- The `SignedSatQ()` and `UnsignedSatQ()` functions when an operation requires, in addition to the saturated result, a Boolean argument that indicates whether saturation occurred.
- The `SignedSat()` and `UnsignedSat()` functions when only the saturated result is required.

`SatQ(i, N, unsigned)` returns either `UnsignedSatQ(i, N)` or `SignedSatQ(i, N)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat(i, N)` or `SignedSat(i, N)` depending on the value of its third argument.

C5.2.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<C> Is an optional field. It specifies the condition under which the instruction is executed. See the section *Conditional execution in Chapter F2 of the ARM ARM* for the range of available conditions and their encoding. If <C> is omitted, it defaults to *always* (AL).

<q> Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

.N Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

.W Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description. The assembler syntax includes a mandatory .W qualifier, along with a note describing the cases in which it applies, where this qualifier is required to select a particular encoding for an instruction. Additional assembler syntax will describe the syntax when the conditions are not met.

———— Note ————

When assembling to the A32 instruction set, the .N qualifier produces an assembler error and the .W qualifier has no effect.

C5.2.3 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table C5-1 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to Floating-point single-precision registers. That is, $D_m[x]$ in a 32-bit context ($0 \leq m \leq 15$, $0 \leq x \leq 1$) is equivalent to $S[2m + x]$.

Table C5-1 Encoding of scalars in multiply instructions

Scalar mnemonic	Usual usage	Scalar size	Register specifier	Index specifier	Accessible registers
< $D_m[x]$ >	Second operand	16-bit	$V_m[2:0]$	M, $V_m[3]$	D0-D7
		32-bit	$V_m[3:0]$	M	D0-D15

C5.2.4 Enabling Advanced SIMD and floating-point support

Software must ensure that the required access to the Advanced SIMD and floating-point features is enabled. Most of those controls are described in the section *Configurable instruction enables and disables, and trap controls in Chapter G1 of the ARM ARM*, and this section:

- Summarizes those controls.
- Provides additional information in the following subsections in *Chapter C5 of the ARM ARM*:
 - *FPEXC control of access to Advanced SIMD and floating-point functionality.*

— *EL0 access to Advanced SIMD and floating-point functionality.*

———— **Note** ————

This section shows the controls when the controlling Exception levels are using AArch32. Similar controls are provided when the Exception levels are using AArch64, and then apply to lower Exception levels that are using AArch32.

The controls of access to floating-point and Advanced SIMD functionality are:

General {CP10, CP11} controls

———— **Note** ————

The naming of these controls reflects the conceptual coprocessors CP10 and CP11.

The {CP10, CP11} controls provide general control of the use of floating-point and Advanced SIMD functionality, as follows:

- CPACR.{cp10, cp11} control access from PE modes other than Hyp mode. These fields have no effect on accesses to CP10 and CP11 from Hyp mode.
- In an implementation that includes EL3, NSACR.{cp10, cp11} control access from Non-secure state.
- In an implementation that includes EL2, if NSACR.{cp10, cp11} permit Non-secure accesses, or if EL3 is not implemented, HCPTR.{TCP10, TCP11} provide an additional control on those accesses.

In each case, the {CP10, CP11} controls must be programmed to the same value, otherwise operation is CONSTRAINED UNPREDICTABLE.

The ARMv8 CONSTRAINED UNPREDICTABLE behavior is that, for all purposes other than reading the value of the register field, behavior is as if the access control field for CP11 has the same value as the access control field for CP10.

Control of accesses to the CPACR from Non-secure PL1 modes

As stated in *General {CP10, CP11} controls*, the CPACR controls access to floating-point and Advanced SIMD functionality from PE modes other than Hyp mode. Accesses to the CPACR from Non-secure PL1 modes can be trapped to EL2.

Additional controls of Advanced SIMD functionality

- If implemented as an RW field, CPACR.ASEDIS can make all Advanced SIMD instructions UNDEFINED in all modes other than Hyp mode.
- In an implementation that includes EL3, when CPACR.ASEDIS permits use of the Advanced SIMD instructions or if the CPACR.ASEDIS control is not implemented, NSACR.NSASEDIS can make all Advanced SIMD instructions UNDEFINED in Non-secure state.
- In an implementation that includes EL2, when the CPACR and NSACR settings permit Non-secure use of the Advanced SIMD instructions, if HCPTR.TASE is implemented as an RW field it can make these instructions UNDEFINED in Hyp mode, and trap to Hyp mode any use of these instructions in a Non-secure PL0 or PL1 mode.

FPEXC control of access to Advanced SIMD and floating-point functionality

In addition, FPEXC.EN is an enable bit for most Advanced SIMD and floating-point operations. When FPEXC.EN is 0, all Advanced SIMD and floating-point instructions are treated as UNDEFINED except for:

- A VMSR to the FPEXC or FPSID register.
- A VMRS from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2 register.

These instructions can be executed only at EL1 or higher.

Note

- When the FPSID is accessible, any write access to the FPSID is ignored.
 - When FPEXC.EN is 0, these operations are treated as UNDEFINED:
 - A VMSR to the FPSCR.
 - A VMRS from the FPSCR.
-

EL0 access to Advanced SIMD and floating-point functionality

When the access controls summarized in this section permit EL0 access to the Advanced SIMD and floating-point functionality, this applies only to the subset of functionality that is available at EL0. In particular:

- Only Advanced SIMD and Floating-point system register that is accessible is the FPSCR.
- The Advanced SIMD and floating-point instructions are available.

Execution at EL0 corresponds to the application level view of the *Advanced SIMD and floating-point functionality*, as described in *Advanced SIMD and floating-point system registers in Chapter E1 of the ARM ARM*.

RETIRED

RETIRED

Chapter C6

AArch32 Register Descriptions

This chapter describes the AArch32 System registers that are added or or affected by ARMv8.1. It contains the following section:

- *General information about AArch32 System registers on page C6-590.*
- *General system control registers on page C6-591.*
- *Debug registers on page C6-644.*
- *Performance Monitors registers on page C6-673.*
- *Generic Timer registers on page C6-687.*
- *ARMv8.0 sections relating to these registers on page C6-698.*

C6.1 General information about AArch32 System registers

The structure of the System register descriptions has changed from that used in *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, Issue A.j and earlier:

- Information about the accessibility of the register from different Exception levels is given in the Accessibility section, that is the last but one section of a register description.
- Information about the traps and enables that apply to the register is given in the Traps and Enables section, that is the last section of a register description.

The information in these sections can depend on the value of one or both of the controls {TGE, NS}.

The TGE control is:

- The [HCR_EL2.TGE](#) field when EL2 is using AArch64.
- The [HCR.TGE](#) field when EL2 is using AArch32.

The NS control is:

- The [SCR_EL3.NS](#) field when EL3 is using AArch64.
- The [SCR.NS](#) field when EL3 is using AArch32.

———— Note ————

- These changes mean the registers descriptions can address:
 - Cases where a single register is accessible using more than one mnemonic, in different contexts, and that the accessibility can depend on the mnemonic used and the context in which it is used.
 - Cases where a single mnemonic can address different registers, depending on the context, and that the accessibility can also depend on the context.

These changes are needed to describe the AArch64 System register behaviors associated with the Virtualization Host Extension described in [Chapter B8 Virtualization Host Extensions](#). However, they also improve the representation of many ARMv8.0 register descriptions, including descriptions of register banking in AArch32 state. Therefore, they apply to both the AArch32 and the AArch64 System register descriptions.

- This change to the structure of System register descriptions does not apply to the description of memory-mapped registers such as those described in [Chapter D2 External Debug Register Descriptions](#).

C6.1.1 The AArch32 register descriptions included in this supplement

For the AArch32 System registers, this supplement includes the full description of all registers that are changed by ARMv8.1, including registers where the only changes introduced by ARMv8.1 are to the accessibility of the register.

The AArch32 System registers descriptions in this chapter do not highlight where ARMv8.1 has changed the register field descriptions. However:

- The field descriptions indicate any differences in behavior between ARMv8.0 and ARMv8.1.
- The descriptions of the features of ARMv8.1 elsewhere in this manual indicate where ARMv8.1 has introduced new register fields, or significantly changed the effect of a register field.

C6.2 General system control registers

This section lists the ARMv8.1 system registers in AArch32 state that are not part of one of the other listed groups.

RETIRED

C6.2.1 CPSR, Current Program Status Register

The CPSR characteristics are:

Purpose

Holds PE status and control information.

Configurations

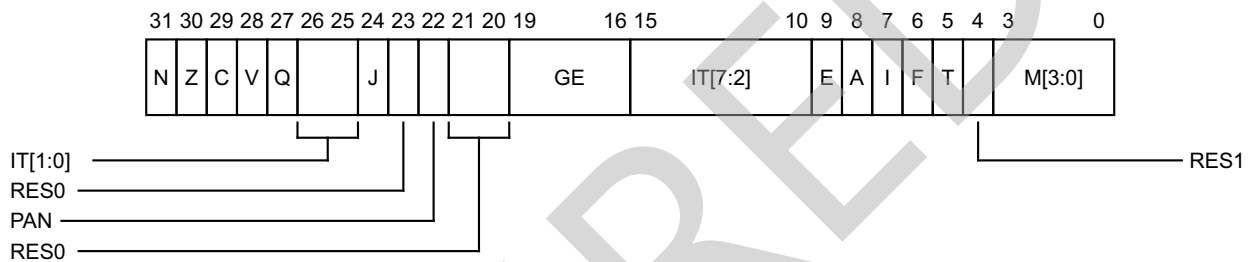
There is one instance of this register that is used in both Secure and Non-secure states.

Attributes

CPSR is a 32-bit register.

Field descriptions

The CPSR bit assignments are:



N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Privileged Access Never. Defined values are:

- 0 The translation system is the same as ARMv8.0.
- 1 Disables privileged read and write accesses.

On taking an exception from the current mode or Exception level, to EL1, EL2 or EL3:

- If the target Exception level is using AArch32, this bit is copied to [SPSR](#).
- If the target is EL1 using AArch64, this bit is copied to [SPSR_EL1](#).
- If the target is EL2 using AArch64, this bit is copied to [SPSR_EL2](#).
- If the target is EL3 using AArch64, this bit is copied to [SPSR_EL2](#).

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the [SCTLR.SPAN](#) bit for the current Security state is 0, this bit is set to 1.
- When the target of the exception is EL3, from Secure state, and the value of the Secure [SCTLR.SPAN](#) is 0, this bit is set to 1.
- When the target of the exception is EL3, from Non-secure state, this bit is set to 0 regardless of the value of the Secure [SCTLR.SPAN](#) bit.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bits [21:20]

Reserved, RES0.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Indicates the AArch32 instruction set state. Possible values of this bit are:

- 0 A32 state.
- 1 T32 state.

Bit [4]

Reserved, RES1.

M[3:0], bits [3:0]

Current PE mode. Possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved.

C6.2.2 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

Purpose

Provides top level information about the debug system in AArch32.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register ID_DFR0 is architecturally mapped to AArch64 System register [ID_DFR0_EL1](#).

Attributes

ID_DFR0 is a 32-bit register.

Field descriptions

The ID_DFR0 bit assignments are:

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0	PerfMon	MProfDbg	MMapTrc	CopTrc	MMapDbg	CopSDBG	CopDbg								

Bits [31:28]

Reserved, RES0.

PerfMon, bits [27:24]

Performance Monitors. Support for System registers-based ARM Performance Monitors Extension, using registers in the coproc == 1111 encoding space, for A and R profile processors. Defined values are:

0000	Performance Monitors Extension system registers not implemented.
0001	Support for Performance Monitors Extension version 1 (PMUv1) System registers.
0010	Support for Performance Monitors Extension version 2 (PMUv2) System registers.
0011	Support for Performance Monitors Extension version 3 (PMUv3) System registers.
0100	Support for Performance Monitors Extension version 3 (PMUv3) System registers, with a 16-bit evtCount field.
1111	IMPLEMENTATION DEFINED form of Performance Monitors System registers supported. PMUv3 not supported.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0011, and 1111.

In ARMv8.1 the permitted values are 0000, 0100, and 1111.

In ARMv7, the value 0000 can mean that PMUv1 is implemented. PMUv1 is not permitted in an ARMv8 implementation.

MProfDbg, bits [23:20]

M Profile Debug. Support for memory-mapped debug model for M profile processors. Defined values are:

0000	Not supported.
0001	Support for M profile Debug architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

MMapTrc, bits [19:16]

Memory Mapped Trace. Support for memory-mapped trace model. Defined values are:

0000 Not supported.

0001 Support for ARM trace architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In the Trace registers, the ETMIDR gives more information about the implementation.

CopTrc, bits [15:12]

Support for System registers-based trace model, using registers in the coproc == 1110 encoding space. Defined values are:

0000 Not supported.

0001 Support for ARM trace architecture, with System registers access.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In the Trace registers, the ETMIDR gives more information about the implementation.

MMapDbg, bits [11:8]

Memory Mapped Debug. Support for v7 memory-mapped debug model, for A and R profile processors.

In ARMv8-A this field is RES0.

The optional memory map defined by ARMv8 is not compatible with ARMv7.

CopSDBG, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = 1110 encoding space, for an A profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-Secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

CopDbg, bits [3:0]

Support for System registers-based debug model, using registers in the coproc == 1110 encoding space, for A and R profile processors. Defined values are:

0000 Not supported.

0010 Support for ARMv6, v6 Debug architecture, with System registers access.

0011 Support for ARMv6, v6.1 Debug architecture, with System registers access.

0100 Support for ARMv7, v7 Debug architecture, with System registers access.

0101 Support for ARMv7, v7.1 Debug architecture, with System registers access.

0110 Support for ARMv8 debug architecture, with System registers access.

0111 Support for ARMv8 debug architecture, with System registers access, and Virtualization Host extensions.

All other values are reserved.

In ARMv8-A the permitted values are 0000, and 0110.

In ARMv8.1 the permitted values are 0000, and 0111.

Accessing the ID_DFR0

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c0, c1, 2	000	010	0000	1111	0001

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c0, c1, 2	x	x	0	-	RO	n/a	RO
p15, 0, <Rt>, c0, c1, 2	x	0	1	-	RO	RO	RO
p15, 0, <Rt>, c0, c1, 2	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TID3](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HSTR_EL2.T0](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TID3](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HSTR_EL2.T0](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and SCR_EL3.NS == 1:

- If [HCR.TID3](#)==1, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.
- If [HSTR.T0](#)==1, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.

C6.2.3 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register ID_ISAR5 is architecturally mapped to AArch64 System register [ID_ISAR5_EL1](#).

Attributes

ID_ISAR5 is a 32-bit register.

Field descriptions

The ID_ISAR5 bit assignments are:

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0	RDM	RES0	CRC32	SHA2	SHA1	AES	SEVL								

Bits [31:28]

Reserved, RES0.

RDM, bits [27:24] (In ARMv8.1)

VQRDMLAH and VQRDMLSH instructions in AArch32. Defined values are:

0000 No VQRDMLAH and VQRDMLSH instructions implemented.

0001 VQRDMLAH and VQRDMLSH instructions implemented.

All other values are reserved.

In ARMv8.0 the only permitted value is 0000.

In ARMv8.1 the only permitted value is 0001.

Bits [27:24] (In ARMv8.0)

Reserved, RES0.

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates whether CRC32 instructions are implemented in AArch32.

0000 No CRC32 instructions implemented.

0001 CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In ARMv8.1 the only permitted value is 0001.

SHA2, bits [15:12]

Indicates whether SHA2 instructions are implemented in AArch32.

0000 No SHA2 instructions implemented.

0001 SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

SHA1, bits [11:8]

Indicates whether SHA1 instructions are implemented in AArch32.

0000 No SHA1 instructions implemented.

0001 SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

AES, bits [7:4]

Indicates whether AES instructions are implemented in AArch32.

0000 No AES instructions implemented.

0001 AESE, AESD, AESMC, and AESIMC implemented.

0010 As for 0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0010.

SEVL, bits [3:0]

Indicates whether the SEVL instruction is implemented in AArch32.

0000 SEVL is implemented as a NOP.

0001 SEVL is implemented as Send Event Local.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

Accessing the ID_ISAR5

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c0, c2, 5	000	101	0000	1111	0010

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c0, c2, 5	x	x	0	-	RO	n/a	RO
p15, 0, <Rt>, c0, c2, 5	x	0	1	-	RO	RO	RO
p15, 0, <Rt>, c0, c2, 5	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page C6-698 for exceptions taken to AArch32 state and *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547 for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and $(SCR_EL3.NS == 1)$ AND $(HCR_EL2.E2H == 0)$:

- If $HCR_EL2.TID3 == 1$, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If $HSTR_EL2.T0 == 1$, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and $(SCR_EL3.NS == 1)$ AND $(HCR_EL2.E2H == 1)$ AND $(HCR_EL2.TGE == 0)$:

- If $HCR_EL2.TID3 == 1$, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If $HSTR_EL2.T0 == 1$, Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and $SCR_EL3.NS == 1$:

- If $HCR.TID3 == 1$, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.
- If $HSTR.T0 == 1$, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.

C6.2.4 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register ID_MMFR3 is architecturally mapped to AArch64 System register [ID_MMFR3_EL1](#).

Attributes

ID_MMFR3 is a 32-bit register.

Field descriptions

The ID_MMFR3 bit assignments are:

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Supersec	CMemSz	CohWalk	PAN	MaintBcst	BPMaint	CMaintSW	CMaintVA								

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported.

Defined values are:

0000 Supersections supported.

1111 Supersections not supported.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 1111.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

0000 4GB, corresponding to a 32-bit physical address range.

0001 64GB, corresponding to a 36-bit physical address range.

0010 1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0001, and 0010.

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the point of unification. Defined values are:

0000 Updates to the translation tables require a clean to the point of unification to ensure visibility by subsequent translation table walks.

0001 Updates to the translation tables do not require a clean to the point of unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

PAN, bits [19:16] (In ARMv8.1)

Privileged Access Never. Indicates support for the PAN bit in [CPSR](#), [SPSR](#), and [DPSR](#) in AArch32. Defined values are:

0000 PAN not supported.

0001 PAN supported.

All other values are reserved.

In ARMv8.1 the only permitted value is 0001.

Bits [19:16] (In ARMv8.0)

Reserved, RES0.

MaintBest, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

0000 Cache, TLB, and branch predictor operations only affect local structures.

0001 Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

0010 Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Defined values are:

0000 None supported.

0001 Supported branch predictor maintenance operations are:

- Invalidate all branch predictors.

0010 As for 0001, and adds:

- Invalidate branch predictors by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance instructions by set/way are:

- Invalidate data cache by set/way.
- Clean data cache by set/way.
- Clean and invalidate data cache by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance operations by VA are:

- Invalidate data cache by VA.
- Clean data cache by VA.
- Clean and invalidate data cache by VA.
- Invalidate instruction cache by VA.
- Invalidate all instruction cache entries.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Accessing the ID_MMFR3

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c0, c1, 7	000	111	0000	1111	0001

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c0, c1, 7	x	x	0	-	RO	n/a	RO
p15, 0, <Rt>, c0, c1, 7	x	0	1	-	RO	RO	RO
p15, 0, <Rt>, c0, c1, 7	x	1	1	-	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TID3==1](#), Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HSTR_EL2.T0==1](#), Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TID3==1](#), Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HSTR_EL2.T0==1](#), Non-secure read accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and $\text{SCR_EL3.NS} = 1$:

- If $\text{HCR.TID3} = 1$, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.
- If $\text{HSTR.T0} = 1$, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.

RETIRED

C6.2.5 SCTLR, System Control Register

The SCTLR characteristics are:

Purpose

Provides the top level control of the system, including its memory system.

Configurations

AArch32 System register SCTLR is architecturally mapped to AArch64 System register [SCTLR_EL1](#).

When EL3 is using AArch32, write access to SCTLR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

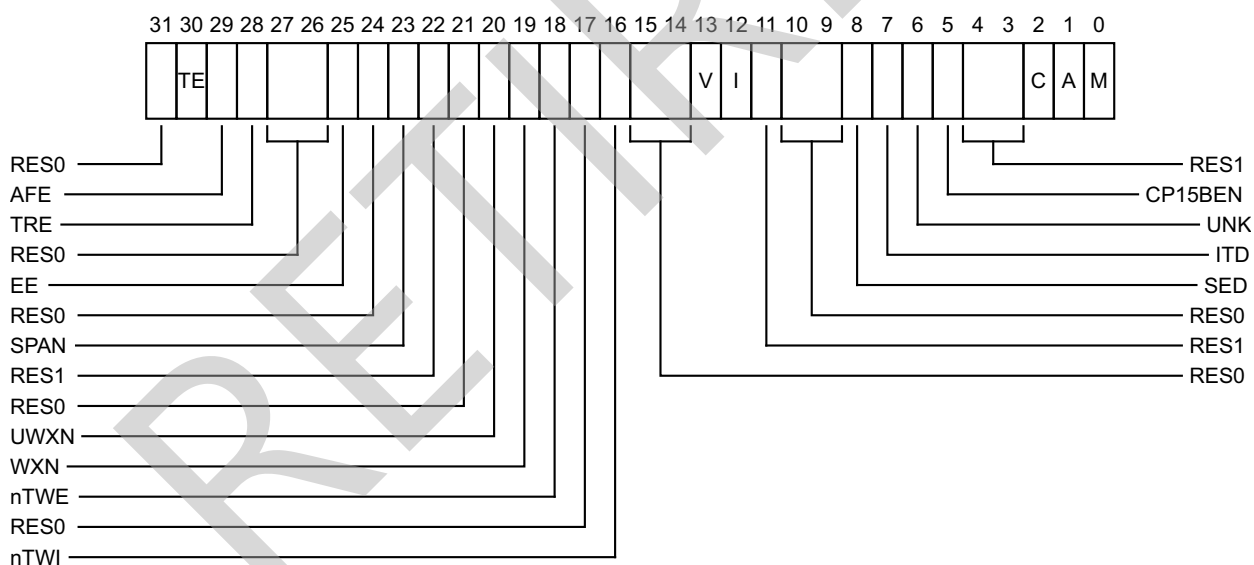
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 they apply only to the Secure instance of the register. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

Attributes

SCTLR is a 32-bit register.

Field descriptions

The SCTLR bit assignments are:



Bit [31]

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to an Exception Level that is executing at PL1 are taken to A32 or T32 state:

- 0 Exceptions, including reset, taken to A32 state.
- 1 Exceptions, including reset, taken to T32 state.

When this register has an architecturally-defined reset value, this field resets to an IMPLEMENTATION DEFINED choice between:

- 0.

- A value determined by an input configuration signal.

AFE, bit [29]

Access Flag Enable. When using the Short-descriptor translation table format for the PL1&0 translation regime, this bit enables use of the AP[0] bit in the translation descriptors as the Access flag, and restricts access permissions in the translation descriptors to the simplified model. The possible values of this bit are:

- | | |
|---|---|
| 0 | In the translation table descriptors, AP[0] is an access permissions bit. The full range of access permissions is supported. No Access flag is implemented. |
| 1 | In the translation table descriptors, AP[0] is the Access flag. Only the simplified model for access permissions is supported. |

When using the Long-descriptor translation table format, the VMSA behaves as if this bit is set to 1, regardless of the value of this bit.

The AFE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

TRE, bit [28]

TEX remap enable. This bit enables remapping of the TEX[2:1] bits in the PL1&0 translation regime for use as two translation table bits that can be managed by the operating system. Enabling this remapping also changes the scheme used to describe the memory region attributes in the VMSA. The possible values of this bit are:

- | | |
|---|--|
| 0 | TEX remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory region attributes. |
| 1 | TEX remap enabled. TEX[2:1] are reassigned for use as bits managed by the operating system. The TEX[0], C, and B bits are used to describe the memory region attributes, with the MMU remap registers. |

When the value of TTBCR.EAE is 1, this bit is RES1.

The TRE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

The value of the PSTATE.E bit on branch to an exception vector or coming out of reset, and the endianness of stage 1 translation table walks in the PL1&0 translation regime.

The possible values of this bit are:

- | | |
|---|---|
| 0 | Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are little-endian. |
| 1 | Big-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are big-endian. |

If an implementation does not provide Big-endian support for data accesses at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support for data accesses at Exception Levels higher than EL0, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to an IMPLEMENTATION DEFINED choice between:

- 0.
- A value determined by an input configuration signal.

Bit [24]

Reserved, RES0.

SPAN, bit [23] (In ARMv8.1)

Set Privileged Access Never, on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

0 CPSR.PAN is set to 1 in the following situations:

- In Non-secure state, on taking an exception to EL1.
- In Secure state, when EL3 is using AArch64, on taking an exception to EL1.
- In Secure state, when EL3 is using AArch32, on taking an exception to EL3.

1 The value of CPSR.PAN is left unchanged on taking an exception.

Bit [23] (In ARMv8.0)

Reserved, RES1.

Bit [22]

Reserved, RES1.

Bit [21]

Reserved, RES0.

UWXN, bit [20]

Unprivileged write permission implies PL1 XN (Execute-never). This bit can force all memory regions that are writeable at PL0 to be treated as XN for accesses from software executing at PL1. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writeable at PL0 forced to XN for accesses from software executing at PL1.

The UWXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the PL1&0 translation regime, this bit can force all memory regions that are writeable to be treated as XN. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writeable in the PL1&0 translation regime is forced to XN for accesses from software executing at PL1 or PL0.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

nTWE, bit [18]

Traps PL0 execution of WFE instructions to Undefined mode.

0 Any attempt to execute a WFE instruction at PL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

1 This control has no effect on the PL0 execution of WFE instructions.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

————— Note —————

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to 1.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps PL0 execution of WFI instructions to Undefined mode.

- 0 Any attempt to execute a WFI instruction at PL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
- 1 This control has no effect on the PL0 execution of WFI instructions.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

———— Note ————

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to 1.

Bits [15:14]

Reserved, RES0.

V, bit [13]

Vectors bit. This bit selects the base address of the exception vectors for exceptions taken to a PE mode other than Monitor mode or Hyp mode:

- 0 Normal exception vectors. Base address is held in VBAR.
- 1 High exception vectors (Hivecs), base address 0xFFFF0000. This base address cannot be remapped.

When this register has an architecturally-defined reset value, this field resets to an IMPLEMENTATION DEFINED choice between:

- 0.
- A value determined by an input configuration signal.

I, bit [12]

Instruction access Cacheability control, for accesses at EL1 and EL0:

- 0 All instruction access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of instruction and unified cache.
If the value of SCTL.R.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
- 1 All instruction access to Normal memory from PL1 and PL0 can be cached at all levels of instruction and unified cache.
If the value of SCTL.R.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

Instruction accesses to Normal memory from Non-secure EL1 and Non-secure EL0 are Cacheable regardless of the value of the SCTL.R.I bit if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at PL0 and PL1.

0 SETEND instruction execution is enabled at PL0 and PL1.

1 SETEND instructions are UNDEFINED at PL0 and PL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at PL1 and PL0.

0 All IT instruction functionality is enabled at PL1 and PL0.

1 Any attempt at PL1 or PL0 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0] != 1000.
- All encodings of the subsequent instruction with the following values for hw1:

11xxxxxxxxxxxx

All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

1011xxxxxxxxxxxx

All instructions in [Miscellaneous 16-bit instructions on page C4-573](#).

10100xxxxxxxxxxx

ADD Rd, PC, #imm

01001xxxxxxxxxxx

LDR Rd, [PC, #imm]

0100x1xxx111xxx

ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

010001xx1xxx111

ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the [SCTLR_EL1](#). If it is not implemented then this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

UNK, bit [6]

Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==1111) encoding space from PL1 and PL0:

- | | |
|---|---|
| 0 | PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED. |
| 1 | PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled. |

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the [SCTLR_EL1](#). If it is not implemented then this bit is RAO/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 1.

Bits [4:3]

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL1 and EL0:

- | | |
|---|--|
| 0 | All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache. |
| 1 | All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, can be cached at all levels of data and unified cache. |

The PE ignores SCTLR.C for Non-secure state and data accesses to Normal memory from EL1 and EL0 are Cacheable if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of [HCR_EL2.DC](#) is 1.

When this register has an architecturally-defined reset value, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at PL1 and PL0:

- | | |
|---|---|
| 0 | Alignment fault checking disabled when executing at PL1 or PL0.
Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed. |
| 1 | Alignment fault checking enabled when executing at PL1 or PL0.
All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception. |

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to 0.

M, bit [0]

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

- | | |
|---|---|
| 0 | EL1 and EL0 stage 1 address translation disabled.
See the SCTLR.I field for the behavior of instruction accesses to Normal memory. |
| 1 | EL1 and EL0 stage 1 address translation enabled. |

In the Non-secure state the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if either:

- EL2 is using AArch32 and the value of HCR.{DC, TGE} is not {0, 0}.
- EL2 is using AArch64 and the value of [HCR_EL2.{DC, TGE}](#) is not {0, 0}.

When this register has an architecturally-defined reset value, this field resets to 0.

Accessing the SCTLR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c1, c0, 0	000	000	0001	1111	0000

Accessibility

The register is accessible in software as follows:

<syntax>	Configuration	Control			Accessibility				Instance
		E2H	TGE	NS	EL0	EL1	EL2	EL3	
p15, 0, <Rt>, c1, c0, 0	EL3 unimplemented	x	x	0	-	RW	n/a	n/a	SCTLR_s
p15, 0, <Rt>, c1, c0, 0	EL3 using AArch64	x	x	0	-	RW	n/a	n/a	SCTLR_s
p15, 0, <Rt>, c1, c0, 0	EL3 using AArch32	x	x	0	-	RW	n/a	RW	SCTLR_s
p15, 0, <Rt>, c1, c0, 0	EL3 unimplemented	x	0	1	-	RW	RW	n/a	SCTLR_ns
p15, 0, <Rt>, c1, c0, 0	EL3 unimplemented	x	1	1	-	n/a	RW	n/a	SCTLR_ns
p15, 0, <Rt>, c1, c0, 0	EL3 using AArch64	x	0	1	-	RW	RW	n/a	SCTLR_ns
p15, 0, <Rt>, c1, c0, 0	EL3 using AArch64	x	1	1	-	n/a	RW	n/a	SCTLR_ns
p15, 0, <Rt>, c1, c0, 0	EL3 using AArch32	x	0	1	-	RW	RW	RW	SCTLR_ns
p15, 0, <Rt>, c1, c0, 0	EL3 using AArch32	x	1	1	-	n/a	RW	RW	SCTLR_ns

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.
- If [HSTR_EL2.T1](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If [HCR_EL2.TVM](#)==1, Non-secure write accesses to this register from EL1 are trapped to EL2.
- If [HCR_EL2.TRVM](#)==1, Non-secure read accesses to this register from EL1 are trapped to EL2.

- If `HSTR_EL2.T1`==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and `SCR_EL3.NS` == 1:

- If `HCR.TVM`==1, Non-secure write accesses to this register from EL1 are trapped to Hyp mode.
- If `HCR.TRVM`==1, Non-secure read accesses to this register from EL1 are trapped to Hyp mode.
- If `HSTR.T1`==1, Non-secure accesses to this register from EL1 are trapped to Hyp mode.

RETIRED

C6.2.6 SPSR, Saved Program Status Register

The SPSR characteristics are:

Purpose

Holds the saved process state for the current mode.

Configurations

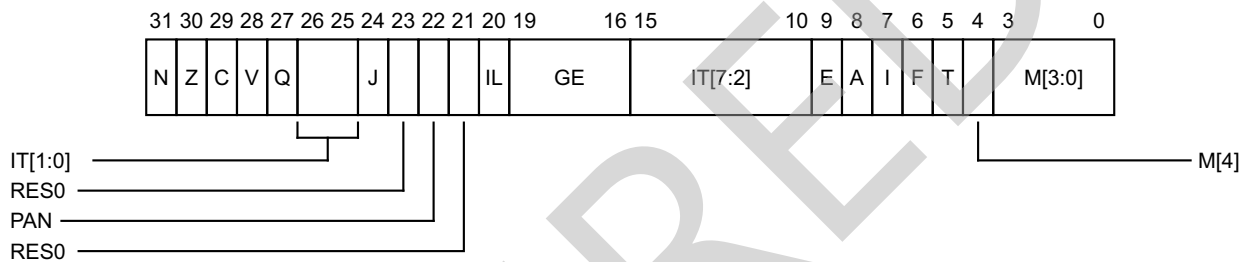
There is one instance of this register that is used in both Secure and Non-secure states.

Attributes

SPSR is a 32-bit register.

Field descriptions

The SPSR bit assignments are:



N, bit [31]

Set to the value of **CPSR.N** on taking an exception to the current mode, and copied to **CPSR.N** on executing an exception return operation in the current mode.

Z, bit [30]

Set to the value of **CPSR.Z** on taking an exception to the current mode, and copied to **CPSR.Z** on executing an exception return operation in the current mode.

C, bit [29]

Set to the value of **CPSR.C** on taking an exception to the current mode, and copied to **CPSR.C** on executing an exception return operation in the current mode.

V, bit [28]

Set to the value of **CPSR.V** on taking an exception to the current mode, and copied to **CPSR.V** on executing an exception return operation in the current mode.

Q, bit [27]

Set to the value of **CPSR.Q** on taking an exception to the current mode, and copied to **CPSR.Q** on executing an exception return operation in the current mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to the current mode, and copied to [CPSR.PAN](#) on executing an exception return operation in the current mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- | | |
|---|-------------------------|
| 0 | Little-endian operation |
| 1 | Big-endian operation. |

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- | | |
|---|-----------------------|
| 0 | Exception not masked. |
| 1 | Exception masked. |

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- | | |
|---|-----------------------|
| 0 | Exception not masked. |
| 1 | Exception masked. |

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved.

C6.2.7 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

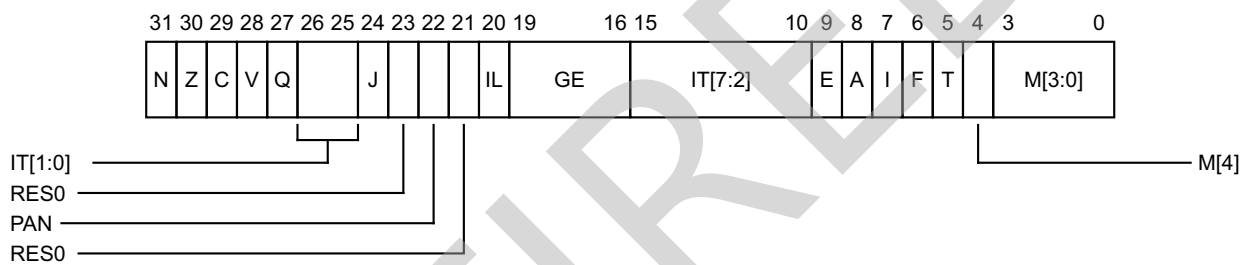
AArch32 System register SPSR_abt is architecturally mapped to AArch64 System register [SPSR_abt](#).

Attributes

SPSR_abt is a 32-bit register.

Field descriptions

The SPSR_abt bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Abort mode, and copied to [CPSR.N](#) on executing an exception return operation in Abort mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Abort mode, and copied to [CPSR.Z](#) on executing an exception return operation in Abort mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Abort mode, and copied to [CPSR.C](#) on executing an exception return operation in Abort mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Abort mode, and copied to [CPSR.V](#) on executing an exception return operation in Abort mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to Abort mode, and copied to [CPSR.Q](#) on executing an exception return operation in Abort mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to Abort mode, and copied to [CPSR.PAN](#) on executing an exception return operation in Abort mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_abt

This register can be read using MRS (banked register) with the following syntax:

```
MRS <Rd>, <banked_reg>
```

This register can be written using MSR (banked register) with the following syntax:

```
MSR <banked_reg>, <Rd>
```

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_abt	1	1	0100

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_abt	x	x	0	-	RW	n/a	RW
SPSR_abt	x	0	1	-	RW	RW	RW
SPSR_abt	x	1	1	-	n/a	RW	RW

This register is only accessible at EL1 in modes other than Abort mode. In Abort mode, it is accessible as the current SPSR.

C6.2.8 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

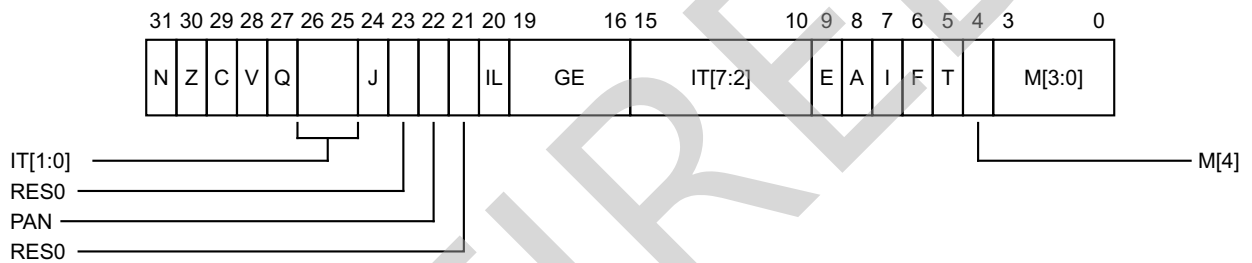
AArch32 System register SPSR_fiq is architecturally mapped to AArch64 System register [SPSR_fiq](#).

Attributes

SPSR_fiq is a 32-bit register.

Field descriptions

The SPSR_fiq bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to FIQ mode, and copied to [CPSR.N](#) on executing an exception return operation in FIQ mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to FIQ mode, and copied to [CPSR.Z](#) on executing an exception return operation in FIQ mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to FIQ mode, and copied to [CPSR.C](#) on executing an exception return operation in FIQ mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to FIQ mode, and copied to [CPSR.V](#) on executing an exception return operation in FIQ mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to FIQ mode, and copied to [CPSR.Q](#) on executing an exception return operation in FIQ mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to FIQ mode, and copied to [CPSR.PAN](#) on executing an exception return operation in FIQ mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_fiq

This register can be read using MRS (banked register) with the following syntax:

```
MRS <Rd>, <banked_reg>
```

This register can be written using MSR (banked register) with the following syntax:

```
MSR <banked_reg>, <Rd>
```

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_fiq	1	0	1110

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_fiq	x	x	0	-	RW	n/a	RW
SPSR_fiq	x	0	1	-	RW	RW	RW
SPSR_fiq	x	1	1	-	n/a	RW	RW

This register is only accessible at EL1 in modes other than FIQ mode. In FIQ mode, it is accessible as the current SPSR.

C6.2.9 SPSR_hyp, Saved Program Status Register (Hyp mode)

The SPSR_hyp characteristics are:

Purpose

Holds the saved process state when an exception is taken to Hyp mode.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

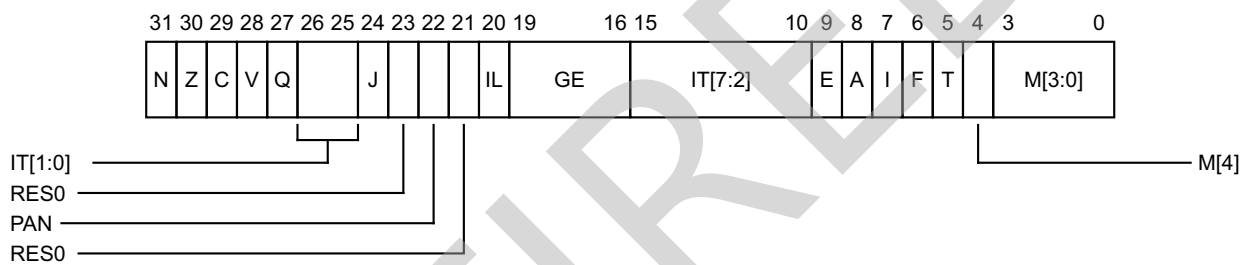
AArch32 System register SPSR_hyp is architecturally mapped to AArch64 System register [SPSR_EL2](#).

Attributes

SPSR_hyp is a 32-bit register.

Field descriptions

The SPSR_hyp bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Hyp mode, and copied to [CPSR.N](#) on executing an exception return operation in Hyp mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Hyp mode, and copied to [CPSR.Z](#) on executing an exception return operation in Hyp mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Hyp mode, and copied to [CPSR.C](#) on executing an exception return operation in Hyp mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Hyp mode, and copied to [CPSR.V](#) on executing an exception return operation in Hyp mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to Hyp mode, and copied to [CPSR.Q](#) on executing an exception return operation in Hyp mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to Hyp mode, and copied to [CPSR.PAN](#) on executing an exception return operation in Hyp mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_hyp

This register can be read using MRS (banked register) with the following syntax:

```
MRS <Rd>, <banked_reg>
```

This register can be written using MSR (banked register) with the following syntax:

```
MSR <banked_reg>, <Rd>
```

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_hyp	1	1	1110

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_hyp	x	x	0	-	-	n/a	RW
SPSR_hyp	x	0	1	-	-	-	RW
SPSR_hyp	x	1	1	-	n/a	-	RW

Note

In Hyp mode, this register is accessible as the current SPSR.

C6.2.10 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

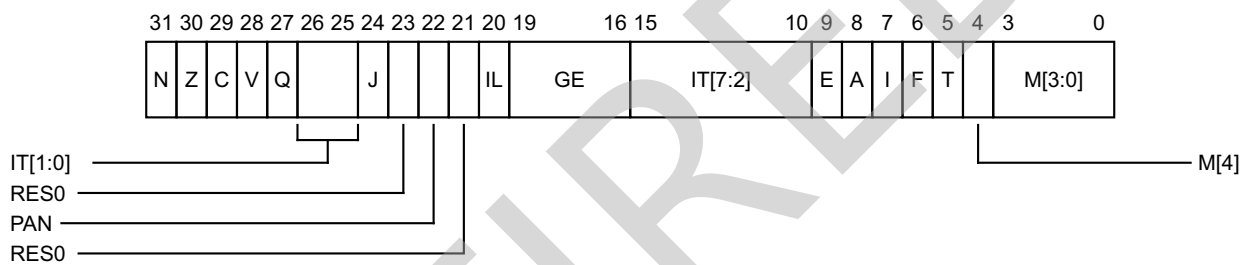
AArch32 System register SPSR_irq is architecturally mapped to AArch64 System register [SPSR_irq](#).

Attributes

SPSR_irq is a 32-bit register.

Field descriptions

The SPSR_irq bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to IRQ mode, and copied to [CPSR.N](#) on executing an exception return operation in IRQ mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to IRQ mode, and copied to [CPSR.Z](#) on executing an exception return operation in IRQ mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to IRQ mode, and copied to [CPSR.C](#) on executing an exception return operation in IRQ mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to IRQ mode, and copied to [CPSR.V](#) on executing an exception return operation in IRQ mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to IRQ mode, and copied to [CPSR.Q](#) on executing an exception return operation in IRQ mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to IRQ mode, and copied to [CPSR.PAN](#) on executing an exception return operation in IRQ mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_irq

This register can be read using MRS (banked register) with the following syntax:

```
MRS <Rd>, <banked_reg>
```

This register can be written using MSR (banked register) with the following syntax:

```
MSR <banked_reg>, <Rd>
```

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_irq	1	1	0000

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_irq	x	x	0	-	RW	n/a	RW
SPSR_irq	x	0	1	-	RW	RW	RW
SPSR_irq	x	1	1	-	n/a	RW	RW

This register is only accessible at EL1 in modes other than IRQ mode. In IRQ mode, it is accessible as the current SPSR.

C6.2.11 SPSR_mon, Saved Program Status Register (Monitor mode)

The SPSR_mon characteristics are:

Purpose

Holds the saved process state when an exception is taken to Monitor mode.

Configurations

This register is only accessible in Secure state.

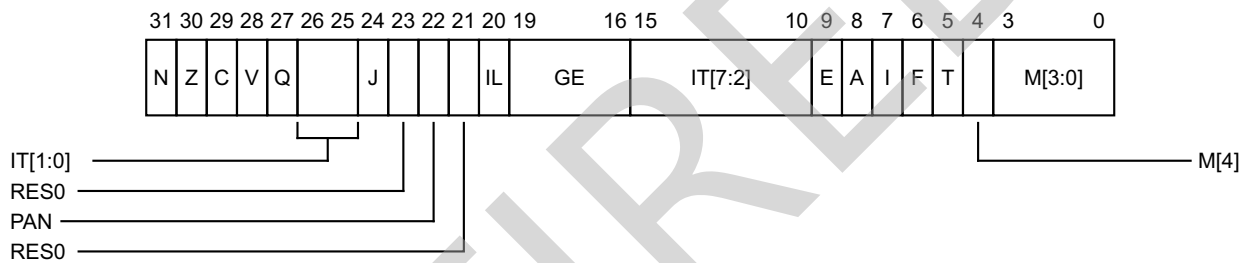
AArch32 System register SPSR_mon can be mapped to AArch64 System register [SPSR_EL3](#), but this is not architecturally mandated.

Attributes

SPSR_mon is a 32-bit register.

Field descriptions

The SPSR_mon bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Monitor mode, and copied to [CPSR.N](#) on executing an exception return operation in Monitor mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Monitor mode, and copied to [CPSR.Z](#) on executing an exception return operation in Monitor mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Monitor mode, and copied to [CPSR.C](#) on executing an exception return operation in Monitor mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Monitor mode, and copied to [CPSR.V](#) on executing an exception return operation in Monitor mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to Monitor mode, and copied to [CPSR.Q](#) on executing an exception return operation in Monitor mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of [CPSR.PAN](#) on taking an exception to Monitor mode, and copied to [CPSR.PAN](#) on executing an exception return operation in Monitor mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of [PSTATE.IL](#) immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the [CPSR.E](#) bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_mon

This register can be read using MRS (banked register) with the following syntax:

MRS <Rd>, <banked_reg>

This register can be written using MSR (banked register) with the following syntax:

MSR <banked_reg>, <Rd>

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_mon	1	1	1100

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_mon	x	x	0	-	-	n/a	RW
SPSR_mon	x	0	1	-	-	-	RW
SPSR_mon	x	1	1	-	n/a	-	RW

This register is only accessible at EL3 in modes other than Monitor mode. In Monitor mode, it is accessible as the current SPSR.

C6.2.12 SPSR_svc, Saved Program Status Register (Supervisor mode)

The SPSR_svc characteristics are:

Purpose

Holds the saved process state when an exception is taken to Supervisor mode.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

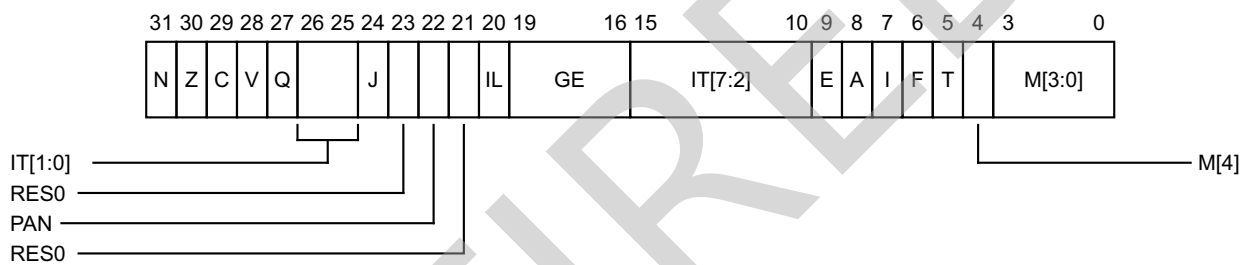
AArch32 System register SPSR_svc is architecturally mapped to AArch64 System register [SPSR_EL1](#).

Attributes

SPSR_svc is a 32-bit register.

Field descriptions

The SPSR_svc bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Supervisor mode, and copied to [CPSR.N](#) on executing an exception return operation in Supervisor mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Supervisor mode, and copied to [CPSR.Z](#) on executing an exception return operation in Supervisor mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Supervisor mode, and copied to [CPSR.C](#) on executing an exception return operation in Supervisor mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Supervisor mode, and copied to [CPSR.V](#) on executing an exception return operation in Supervisor mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to Supervisor mode, and copied to [CPSR.Q](#) on executing an exception return operation in Supervisor mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of **CPSR.PAN** on taking an exception to Supervisor mode, and copied to **CPSR.PAN** on executing an exception return operation in Supervisor mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of **PSTATE.IL** immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- **IT[7:5]** holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- **IT[4:0]** encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is **0b00000000** when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the **SCTLR.EE** bit is defined by a configuration input signal, that value also applies to the **CPSR.E** bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_svc

This register can be read using MRS (banked register) with the following syntax:

```
MRS <Rd>, <banked_reg>
```

This register can be written using MSR (banked register) with the following syntax:

```
MSR <banked_reg>, <Rd>
```

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_svc	1	1	0010

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_svc	x	x	0	-	RW	n/a	RW
SPSR_svc	x	0	1	-	RW	RW	RW
SPSR_svc	x	1	1	-	n/a	RW	RW

This register is only accessible at EL1 in modes other than Supervisor mode. In Supervisor mode, it is accessible as the current SPSR.

C6.2.13 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

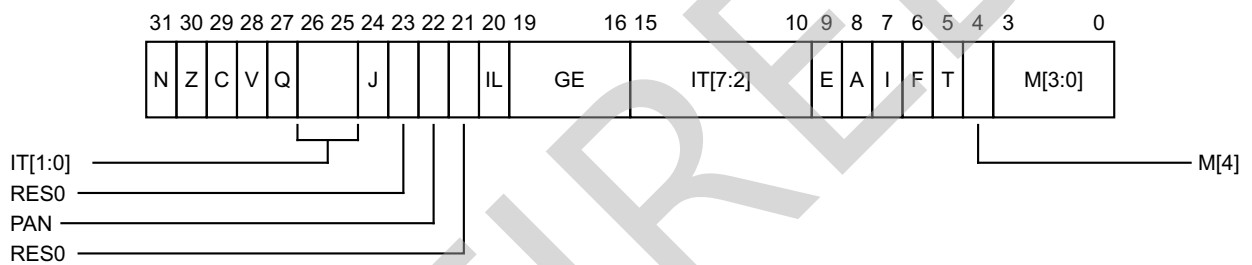
AArch32 System register SPSR_und is architecturally mapped to AArch64 System register [SPSR_und](#).

Attributes

SPSR_und is a 32-bit register.

Field descriptions

The SPSR_und bit assignments are:



N, bit [31]

Set to the value of [CPSR.N](#) on taking an exception to Undefined mode, and copied to [CPSR.N](#) on executing an exception return operation in Undefined mode.

Z, bit [30]

Set to the value of [CPSR.Z](#) on taking an exception to Undefined mode, and copied to [CPSR.Z](#) on executing an exception return operation in Undefined mode.

C, bit [29]

Set to the value of [CPSR.C](#) on taking an exception to Undefined mode, and copied to [CPSR.C](#) on executing an exception return operation in Undefined mode.

V, bit [28]

Set to the value of [CPSR.V](#) on taking an exception to Undefined mode, and copied to [CPSR.V](#) on executing an exception return operation in Undefined mode.

Q, bit [27]

Set to the value of [CPSR.Q](#) on taking an exception to Undefined mode, and copied to [CPSR.Q](#) on executing an exception return operation in Undefined mode.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of **CPSR.PAN** on taking an exception to Undefined mode, and copied to **CPSR.PAN** on executing an exception return operation in Undefined mode.

Bit [22] (In ARMv8.0)

Reserved, RES0.

Bit [21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state bit. Shows the value of **PSTATE.IL** immediately before the exception was taken.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the **SCTLR.EE** bit is defined by a configuration input signal, that value also applies to the **CPSR.E** bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

- 0 Taken from A32 state.
- 1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

- 1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that an exception was taken from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the SPSR_und

This register can be read using MRS (banked register) with the following syntax:

```
MRS <Rd>, <banked_reg>
```

This register can be written using MSR (banked register) with the following syntax:

```
MSR <banked_reg>, <Rd>
```

This syntax is encoded with the following settings in the instruction encoding:

<banked_reg>	R	M	M1
SPSR_und	1	1	0110

Accessibility

The register is accessible in software as follows:

<banked_reg>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
SPSR_und	x	x	0	-	RW	n/a	RW
SPSR_und	x	0	1	-	RW	RW	RW
SPSR_und	x	1	1	-	n/a	RW	RW

This register is only accessible at EL1 in modes other than Undefined mode. In Undefined mode, it is accessible as the current SPSR.

C6.3 Debug registers

This section lists the ARMv8.1 Debug System registers in AArch32 state, in alphabetic order.

———— **Note** ————

The changes to the breakpoint types introduced by Virtualization Host Extensions apply to the AArch32 registers [DBG BXVR<n>](#) and [DBG BVR<n>](#) because matching against [CONTEXTIDR_EL2](#) depends on whether EL2 is using AArch64, and not on the Execution state of the debug target Exception level.

RETIRED

C6.3.1 DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n> characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register [DBGBVR<n>](#). If EL2 is implemented and this breakpoint supports Context matching, [DBGBVR<n>](#) can be associated with a Breakpoint Extended Value Register [DBGBXVR<n>](#) for VMID matching.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGBCR<n> is architecturally mapped to AArch64 System register [DBGBCR<n>_EL1](#).

AArch32 System register DBGBCR<n> is architecturally mapped to External register [DBGBCR<n>_EL1](#).

If breakpoint n is not implemented then this register is unallocated.

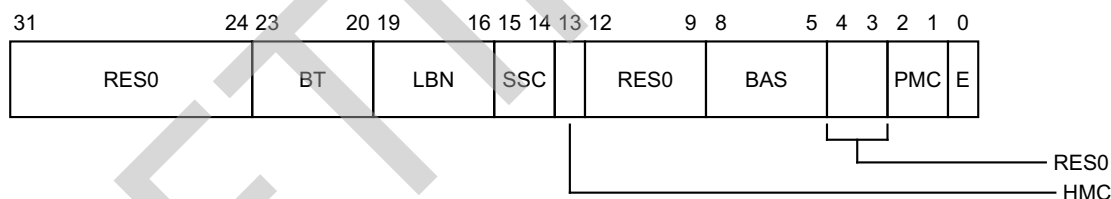
This register is in the Cold reset domain. On a Cold reset RW fields in this register reset to architecturally UNKNOWN values. The register is not affected by a Warm reset.

Attributes

DBGBCR<n> is a 32-bit register.

Field descriptions

The DBGBCR<n> bit assignments are:



When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

0000	Unlinked address match.
0001	Linked address match.
0010	Unlinked Context ID match.
0011	Linked Context ID match.
0100	Unlinked instruction address mismatch.
0101	Linked instruction address mismatch.
0110	Unlinked CONTEXTIDR_EL1 match (ARMv8.1).
0111	Linked CONTEXTIDR_EL1 match (ARMv8.1).
1000	Unlinked VMID match.
1001	Linked VMID match.

1010	Unlinked VMID and Context ID match.
1011	Linked VMID and Context ID match.
1100	Unlinked CONTEXTIDR_EL2 match (ARMv8.1).
1101	Linked CONTEXTIDR_EL2 match (ARMv8.1).
1110	Unlinked Full Context ID match (ARMv8.1).
1111	Linked Full Context ID match (ARMv8.1).

The field breaks down as follows:

- BT[3:1]: Base type.

000	Match address. DBGBVR<n> is the address of an instruction.
001	Match Context ID. DBGBVR<n> .ContextID is a Context ID compared against CONTEXTIDR in ARMv8.0, and in ARMv8.1 when not in a Host OS or a Host Application. In ARMv8.1, when in a Host OS or Host Application, the Context ID is compared against CONTEXTIDR_EL1 .
010	Mismatch address. DBGBVR<n> is the address of an instruction to be stepped.
011	Match CONTEXTIDR_EL1 . DBGBVR<n> .ContextID is a Context ID compared against CONTEXTIDR .
100	Match VMID. DBGBXVR<n> .VMID is a VMID compared against VTTBR.VMID.
101	Match VMID and Context ID. DBGBVR<n> .ContextID is a Context ID compared against CONTEXTIDR , and DBGBXVR<n> .VMID is a VMID compared against VTTBR.VMID.
110	Match CONTEXTIDR_EL2 . DBGBXVR<n> .ContextID2 is a Context ID compared against CONTEXTIDR_EL2 .
111	Match Full Context ID. DBGBVR<n> .ContextID is compared against CONTEXTIDR_EL1 , and DBGBXVR<n> .ContextID2 is compared against CONTEXTIDR_EL2 .
- BT[0]: Enable linking.

All other values are reserved. Constraints on breakpoint programming mean other values are reserved under some conditions. For more information, including the effect of programming this field to a reserved value, see “Reserved [DBGBCR<n>](#).BT values” in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of [DBGBCR<n>](#).E is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see [Usage constraints on page C6-700](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint *n* is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0011	DBGBVR< <i>n</i> >	Use for T32 instructions.
1100	DBGBVR< <i>n</i> >+2	Use for T32 instructions.
1111	DBGBVR< <i>n</i> >	Use for A32 instructions.

All other values are reserved. For more information, see "Reserved DBGBCR<*n*>_EL1.BAS values" in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

For more information on using the BAS field in Address Match breakpoints, see [Using the BAS field in Address Match breakpoints on page C6-700](#).

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Step instruction at	Constraint for debuggers
0000	-	Use for a match anywhere breakpoint.
0011	DBGBVR< <i>n</i> >	Use for stepping T32 instructions.
1100	DBGBVR< <i>n</i> >+2	Use for stepping T32 instructions.
1111	DBGBVR< <i>n</i> >	Use for stepping A32 instructions.

All other values are reserved. For more information, see "Reserved DBGBCR<*n*>_EL1.BAS values" in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

For more information on using the BAS field in address mismatch breakpoints, see [Using the BAS field in Address Match breakpoints on page C6-700](#).

For Context matching breakpoints, this field is RES1 and ignored.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR< n >.SSC description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

E, bit [0]

Enable breakpoint DBGBVR< n >. Possible values are:

- 0 Breakpoint disabled.
- 1 Breakpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the DBGBCR< n >

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p14, 0, <Rt>, c0, <CRm>, 5	000	101	0000	1110	n<3:0>

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p14, 0, <Rt>, c0, <CRm>, 5	x	x	0	-	RW	n/a	RW
p14, 0, <Rt>, c0, <CRm>, 5	x	0	1	-	RW	RW	RW
p14, 0, <Rt>, c0, <CRm>, 5	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If **EDSCR.TDA**==1, and **DBGOSLSR.OSLK**==0, accesses to this register from PL1 and PL2 are trapped to Debug state.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If MDCR_EL2.TDA==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and SCR_EL3.NS == 1:

- If HDCR.TDA==1, Non-secure accesses to this register from EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TDA==1, accesses to this register from PL1 and PL2 are trapped to EL3.

RETIRED

C6.3.2 DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, either the virtual address of an instruction or a context ID. Forms breakpoint n together with control register [DBGBCR<n>](#). If EL2 is implemented and this breakpoint supports Context matching, [DBGBVR<n>](#) can be associated with a Breakpoint Extended Value Register [DBGBXVR<n>](#) for VMID matching.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGBVR<n> is architecturally mapped to AArch64 System register DBGBVR<n> EL1[31:0].

AArch32 System register DBGBVR<n> is architecturally mapped to External register DBGBVR<n>_EL1[31:0].

If breakpoint n is not implemented then this register is unallocated.

This register is in the Cold reset domain. On a Cold reset RW fields in this register reset to architecturally UNKNOWN values. The register is not affected by a Warm reset.

Attributes

How this register is interpreted depends on the value of `DBGBCR<n>.BT`.

- When **DBGBCR<n>.BT** is 0b0x0x, this register holds a virtual address.
- When **DBGBCR<n>.BT** is 0b001x, 0b101x, or 0b111x, this register holds a Context ID.

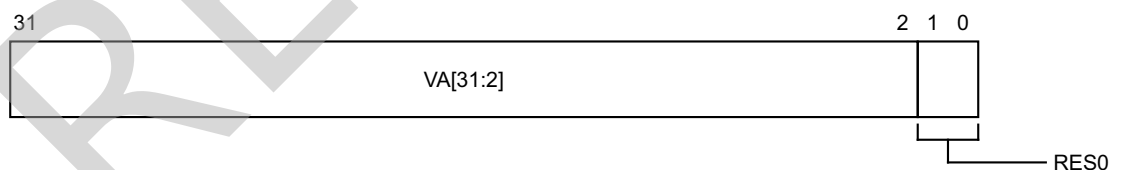
For other values of DBGBCR<n>.BT, this register is RES0.

Some breakpoints might not support Context ID comparison. For more information, see the description of the [DBGDIDR.CTX_CMPs](#) field.

Field descriptions

The DBGBVR<n> bit assignments are:

When `DBGBCR<n>.BT==0b0x0x`:

**VA[31:2], bits [31:2]**

Bits[31:2] of the address value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [1:0]

Reserved, RES0.

When $DBGBCR<n>.BT==0b001x$:



ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR in the following cases:

- The PE is in Secure state.
- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.
- In ARMv8.1, when EL2 is using AArch64, and [HCR_EL2.E2H](#) is 0.
- In ARMv8.1, when EL2 is using AArch64, [HCR_EL2](#).{E2H, TGE} is {1, 0}, and the PE is in Non-secure EL0 or EL1.

In ARMv8.1, when EL2 is using AArch64, [HCR_EL2](#).{E2H, TGE} is {1, 1} and the PE is in Non-secure EL0, the value is compared against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>.BT==0b101x$ and EL2 implemented:



ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>.BT==0b111x$ and EL2 implemented:



ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the $DBGBVR<n>$

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p14, 0, <Rt>, c0, <CRm>, 4	000	100	0000	1110	n<3:0>

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p14, 0, <Rt>, c0, <CRm>, 4	x	x	0	-	RW	n/a	RW
p14, 0, <Rt>, c0, <CRm>, 4	x	0	1	-	RW	RW	RW
p14, 0, <Rt>, c0, <CRm>, 4	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If [EDSCR.TDA](#)==1, and [DBGOSLSR.OSLK](#)==0, accesses to this register from PL1 and PL2 are trapped to Debug state.

When EL2 is implemented and is using AArch64 and [SCR_EL3.NS](#) == 1:

- If [MDCR_EL2.TDA](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and [SCR_EL3.NS](#) == 1:

- If [HDCR.TDA](#)==1, Non-secure accesses to this register from EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If [MDCR_EL3.TDA](#)==1, accesses to this register from PL1 and PL2 are trapped to EL3.

C6.3.3 DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15

The DBGBXVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, to support VMID matching. Used in conjunction with a control register [DBGBCR<n>](#) and a value register [DBGBVR<n>](#), where EL2 is implemented and breakpoint n supports Context matching.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGBXVR<n> is architecturally mapped to AArch64 System register [DBGBVR<n>_EL1](#)[63:32].

AArch32 System register DBGBXVR<n> is architecturally mapped to External register [DBGBVR<n>_EL1](#)[63:32].

This register is unallocated in any of the following cases:

- Breakpoint n is not implemented.
- Breakpoint n does not support Context matching.
- EL2 is not implemented.

For more information, see the description of the [DBGDIDR.CTX_CMPs](#) field.

This register is in the Cold reset domain. On a Cold reset RW fields in this register reset to architecturally UNKNOWN values. The register is not affected by a Warm reset.

Attributes

How this register is interpreted depends on the value of [DBGBCR<n>.BT](#).

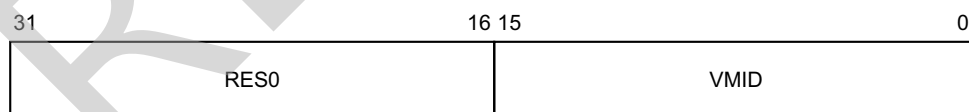
- When [DBGBCR<n>.BT](#) is 0b10xx, this register holds a VMID.
- When [DBGBCR<n>.BT](#) is 0b11xx, this register holds a Context ID.

For other values of [DBGBCR<n>.BT](#), this register is RES0.

Field descriptions

The DBGBXVR<n> bit assignments are:

When [DBGBCR<n>.BT](#)==0b10xx and EL2 implemented:



Bits [31:16]

Reserved, RES0.

VMID, bits [15:0] (In ARMv8.1)

VMID value for comparison.

The VMID is 8 bits in the following cases.

- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.

In ARMv8.1 when EL2 is using AArch64, it is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

The upper 8 bits of this field are RES0 if any of the following apply:

- The implementation has an 8 bit VMID.
- [VTCCR_EL2.VS](#) is 0.
- EL2 is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [15:8] (In ARMv8.0)

Reserved, RES0.

VMID, bits [7:0] (In ARMv8.0)

VMID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When *DBGBCR<n>.BT==0b11xx* and *EL2* implemented:



ContextID2, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the *DBGXVR<n>*

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p14, 0, <Rt>, c1, <CRm>, 1	000	001	0001	1110	n<3:0>

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p14, 0, <Rt>, c1, <CRm>, 1	x	x	0	-	RW	n/a	RW
p14, 0, <Rt>, c1, <CRm>, 1	x	0	1	-	RW	RW	RW
p14, 0, <Rt>, c1, <CRm>, 1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If [EDSCR.TDA](#)==1, and [DBGOSLSR.OSLK](#)==0, accesses to this register from PL1 and PL2 are trapped to Debug state.

When EL2 is implemented and is using AArch64 and [SCR_EL3.NS](#) == 1:

- If [MDCR_EL2.TDA](#)==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and [SCR_EL3.NS](#) == 1:

- If [HDCR.TDA](#)==1, Non-secure accesses to this register from EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If [MDCR_EL3.TDA](#)==1, accesses to this register from PL1 and PL2 are trapped to EL3.

C6.3.4 DBGDIDR, Debug ID Register

The DBGDIDR characteristics are:

Purpose

Specifies which version of the Debug architecture is implemented, and some features of the debug implementation.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

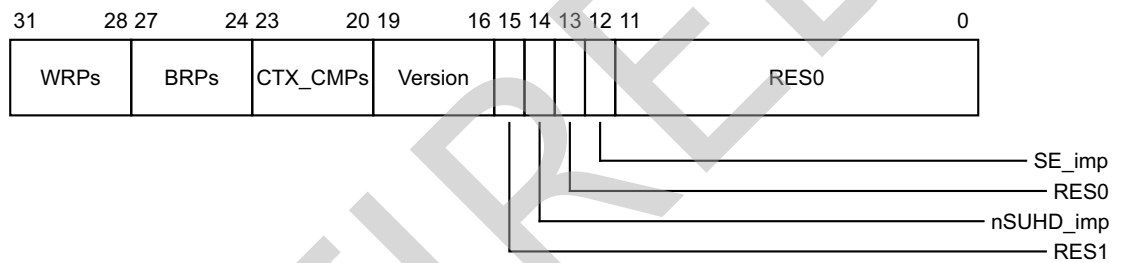
If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

Attributes

DBGDIDR is a 32-bit register.

Field descriptions

The DBGDIDR bit assignments are:



WRPs, bits [31:28]

The number of watchpoints implemented, minus 1.

Permitted values of this field are from 0b0001 for 2 implemented watchpoints, to 0b1111 for 16 implemented watchpoints.

The value of 0b0000 is reserved.

If AArch64 is implemented, this field has the same value as [ID_AA64DFR0_EL1.WRPs](#).

BRPs, bits [27:24]

The number of breakpoints implemented, minus 1.

Permitted values of this field are from 0b0001 for 2 implemented breakpoint, to 0b1111 for 16 implemented breakpoints.

The value of 0b0000 is reserved.

If AArch64 is implemented, this field has the same value as [ID_AA64DFR0_EL1.BRPs](#).

CTX_CMPs, bits [23:20]

The number of breakpoints that can be used for Context matching, minus 1.

Permitted values of this field are from 0b0000 for 1 Context matching breakpoint, to 0b1111 for 16 Context matching breakpoints.

The Context matching breakpoints must be the highest addressed breakpoints. For example, if six breakpoints are implemented and two are Context matching breakpoints, they must be breakpoints 4 and 5.

If AArch64 is implemented, this field has the same value as [ID_AA64DFR0_EL1.CTX_CMPs](#).

Version, bits [19:16]

The Debug architecture version. Defined values are:

0001	ARMv6, v6 Debug architecture.
0010	ARMv6, v6.1 Debug architecture.
0011	ARMv7, v7 Debug architecture, with baseline CP14 registers implemented.
0100	ARMv7, v7 Debug architecture, with all CP14 registers implemented.
0101	ARMv7, v7.1 Debug architecture.
0110	ARMv8, v8 Debug architecture.
0111	ARMv8.1, v8 Debug architecture, with Virtualization Host Extensions.

All other values are reserved.

In ARMv8-A the only permitted value is 0110.

In ARMv8.1 the only permitted value is 0111.

Bit [15]

Reserved, RES1.

nSUHD_imp, bit [14]

In ARMv7-A, was Secure User Halting Debug not implemented.

The value of this bit must match the value of the SE_imp bit.

Bit [13]

Reserved, RES0.

SE_imp, bit [12]

EL3 implemented. The meanings of the values of this bit are:

0	EL3 not implemented.
1	EL3 implemented.

The value of this bit must match the value of the nSUHD_imp bit.

Bits [11:0]

Reserved, RES0.

Accessing the DBGDIDR

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p14, 0, <Rt>, c0, c0, 0	000	000	0000	1110	0000

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p14, 0, <Rt>, c0, c0, 0	x	x	0	RO	RO	n/a	RO
p14, 0, <Rt>, c0, c0, 0	x	0	1	RO	RO	RO	RO
p14, 0, <Rt>, c0, c0, 0	x	1	1	RO	n/a	RO	RO

ARM deprecates any access to this register from EL0.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If [DBGDSCRExt.UDCCdis](#)==1, read accesses to this register from PL0 are trapped to Undefined mode.
- If [MDSCR_EL1.TDCC](#)==1, read accesses to this register from PL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [MDCR_EL2.TDA](#)==1, Non-secure read accesses to this register from EL0 and EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and SCR_EL3.NS == 1:

- If [HDCR.TDA](#)==1, Non-secure read accesses to this register from EL0 and EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If [MDCR_EL3.TDA](#)==1, read accesses to this register from PL0, PL1, and PL2 are trapped to EL3.

C6.3.5 DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

Purpose

Main control register for the debug implementation.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGDSCRext is architecturally mapped to AArch64 System register [MDSCR_EL1](#).

This register is required in all implementations.

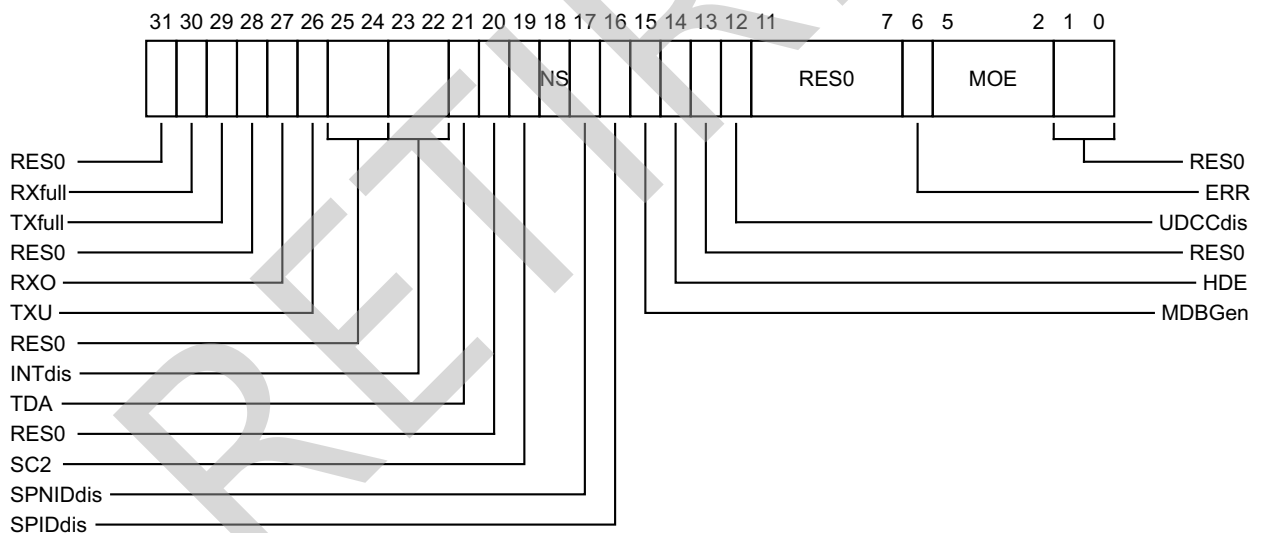
This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset these apply only if the PE resets into an Exception level that is using AArch32. Otherwise, on a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

DBGDSCRext is a 32-bit register.

Field descriptions

The DBGDSCRext bit assignments are:



Bit [31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Used for save/restore of [EDSCR.RXfull](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.RXfull](#).

ARM deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRRX full status.

The architected behavior of this field determines the value it returns after a reset.

TXfull, bit [29]

DTRTX full. Used for save/restore of [EDSCR.TXfull](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.TXfull](#).

ARM deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRTX full status.

The architected behavior of this field determines the value it returns after a reset.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of [EDSCR.RXO](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.RXO](#).

The architected behavior of this field determines the value it returns after a reset.

TXU, bit [26]

Used for save/restore of [EDSCR.TXU](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.TXU](#).

The architected behavior of this field determines the value it returns after a reset.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of [EDSCR.INTdis](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this field is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this field is RW and holds the value of [EDSCR.INTdis](#).

The architected behavior of this field determines the value it returns after a reset.

TDA, bit [21]

Used for save/restore of [EDSCR.TDA](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.TDA](#).

The architected behavior of this field determines the value it returns after a reset.

Bit [20]

Reserved, RES0.

SC2, bit [19] (In ARMv8.1)

Used for save/restore of [EDSCR.SC2](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.SC2](#).

Bit [19] (In ARMv8.0)

Reserved, RES0.

NS, bit [18]

Non-secure status. Returns the inverse of IsSecure(). This bit is RO.

ARM deprecates use of this field.

SPNIDdis, bit [17]

Secure privileged profiling disabled status bit. This bit is RO and reflects the value of ProfilingProhibited(TRUE,EL1). Permitted values are:

- 0 Profiling allowed in Secure privileged modes.
- 1 Profiling prohibited in Secure privileged modes.

ARM deprecates use of this field.

SPIDdis, bit [16]

Secure privileged AArch32 invasive self-hosted debug disabled status bit. This bit is RO and depends on the value of SDCR.SPD and the pseudocode function AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled(). Permitted values are:

- 0 Self-hosted debug enabled in Secure privileged AArch32 modes.
- 1 Self-hosted debug disabled in Secure privileged AArch32 modes.

This bit reads as 1 if any of the following is true and reads as 0 otherwise:

- SDCR.SPD has the value 10.
- SDCR.SPD has the value 00 and AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled() returns FALSE.

ARM deprecates use of this field.

MDBGen, bit [15]

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector Catch debug exceptions.

- 0 Breakpoint, Watchpoint, and Vector Catch debug exceptions disabled.
- 1 Breakpoint, Watchpoint, and Vector Catch debug exceptions enabled.

When this register has an architecturally-defined reset value, this field resets to 0.

HDE, bit [14]

Used for save/restore of [EDSCR.HDE](#).

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.HDE](#).

The architected behavior of this field determines the value it returns after a reset.

Bit [13]

Reserved, RES0.

UDCCdis, bit [12]

Traps PL0 accesses to the DCC registers to Undefined mode.

- 0 PL0 accesses to the DCC registers are not trapped to Undefined mode.

1 PL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, [DBGDIDR](#), DBGDSAR, and DBGDRAR are trapped to Undefined mode.

Note

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

Traps of PL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.
When this register has an architecturally-defined reset value, this field resets to 0.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of [EDSCR.ERR](#).
When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of [EDSCR.ERR](#).
The architected behavior of this field determines the value it returns after a reset.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

0001	Breakpoint
0011	Software breakpoint (BKPT) instruction
0101	Vector catch
1010	Watchpoint

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [1:0]

Reserved, RES0.

Accessing the DBGDSCRext

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p14, 0, <Rt>, c0, c2, 2	000	010	0000	1110	0010

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p14, 0, <Rt>, c0, c2, 2	x	x	0	-	RW	n/a	RW
p14, 0, <Rt>, c0, c2, 2	x	0	1	-	RW	RW	RW
p14, 0, <Rt>, c0, c2, 2	x	1	1	-	n/a	RW	RW

Individual fields within this register might have restricted accessibility when `DBGOSLSR.OSLK == 0` (the OS lock is unlocked.) See the field descriptions for more detail.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and `SCR_EL3.NS == 1`:

- If `MDCR_EL2.TDA==1`, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and `SCR_EL3.NS == 1`:

- If `HDCR.TDA==1`, Non-secure accesses to this register from EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If `MDCR_EL3.TDA==1`, accesses to this register from PL1 and PL2 are trapped to EL3.

C6.3.6 DSPSR, Debug Saved Program Status Register

The DSPSR characteristics are:

Purpose

Holds the saved process state on entry to Debug state.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DSPSR is architecturally mapped to AArch64 System register [DSPSR_EL0](#).

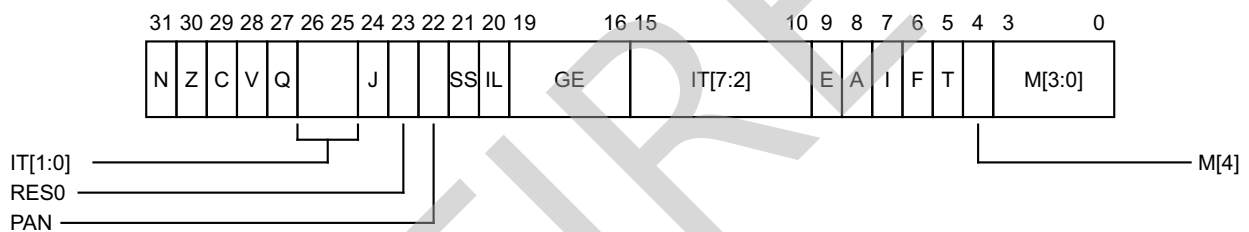
Attributes

DSPSR is a 32-bit register.

Field descriptions

The DSPSR bit assignments are:

When entering Debug state from AArch32 and exiting Debug state to AArch32:



N, bit [31]

Set to the value of [CPSR.N](#) on entering Debug state, and copied to [CPSR.N](#) on exiting Debug state.

Z, bit [30]

Set to the value of [CPSR.Z](#) on entering Debug state, and copied to [CPSR.Z](#) on exiting Debug state.

C, bit [29]

Set to the value of [CPSR.C](#) on entering Debug state, and copied to [CPSR.C](#) on exiting Debug state.

V, bit [28]

Set to the value of [CPSR.V](#) on entering Debug state, and copied to [CPSR.V](#) on exiting Debug state.

Q, bit [27]

Set to the value of [CPSR.Q](#) on entering Debug state, and copied to [CPSR.Q](#) on exiting Debug state.

IT[1:0], bits [26:25]

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bit [23]

Reserved, RES0.

PAN, bit [22] (In ARMv8.1)

Set to the value of **CPSR.PAN** on entering Debug state, and copied to **CPSR.PAN** on exiting Debug state.

Bit [22] (In ARMv8.0)

Reserved, RES0.

SS, bit [21]

Software step. Shows the value of **PSTATE.SS** immediately before Debug state was entered.

IL, bit [20]

Illegal Execution state bit. Shows the value of **PSTATE.IL** immediately before Debug state was entered.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is 0b00000000 when no IT block is active.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

- 0 Little-endian operation
- 1 Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the **SCTLR.EE** bit is defined by a configuration input signal, that value also applies to the **CPSR.E** bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

- 0 Exception not masked.
- 1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

- 0 Exception not masked.

1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the Debug state entry was taken from. Possible values of this bit are:

0 Taken from A32 state.
1 Taken from T32 state.

M[4], bit [4]

Execution state that Debug state was entered from. Possible values of this bit are:

1 Exception taken from AArch32.

M[3:0], bits [3:0]

AArch32 mode that Debug state was entered from. The possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32)
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in [Reserved values in System and memory-mapped registers and translation table entries on page C6-700](#).

Accessing the DSPSR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 3, <Rt>, c4, c5, 0	011	000	0100	1111	0101

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 3, <Rt>, c4, c5, 0	x	x	0	RW	RW	n/a	RW
p15, 3, <Rt>, c4, c5, 0	x	0	1	RW	RW	RW	RW
p15, 3, <Rt>, c4, c5, 0	x	1	1	RW	n/a	RW	RW

Access to this register is from Debug state only. During normal execution this register is unallocated.

RETIRED

C6.3.7 HDCR, Hyp Debug Control Register

The HDCR characteristics are:

Purpose

Controls the trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors Extension.

Configurations

AArch32 System register HDCR is architecturally mapped to AArch64 System register [MDCR_EL2](#).

If EL2 is not implemented, this register is RES0 from EL3.

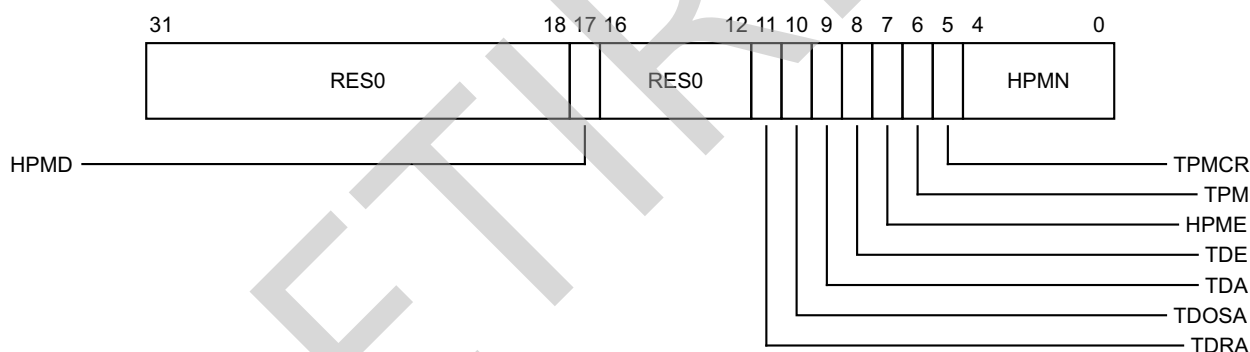
This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset these apply only if the PE resets into an Exception level that is using AArch32. Otherwise, on a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

HDCR is a 32-bit register.

Field descriptions

The HDCR bit assignments are:



Bits [31:18]

Reserved, RES0.

HPMD, bit [17] (In ARMv8.1)

Guest Performance Monitors Disable. This control prohibits event counting at EL2. Permitted values are:

- 0 Event counting allowed in Hyp mode.
- 1 Event counting prohibited in Hyp mode, unless enabled by the IMPLEMENTATION DEFINED authentication interface `ExternalSecureNoninvasiveDebugEnabled()`.

This control applies only to:

- The event counters in the range [0..HPMN).
- If [PMCR](#).DP is set to 1, PMCCNTR.

The other event counters are unaffected, and when [PMCR](#).DP is set to 0, PMCCNTR is unaffected.

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [17] (In ARMv8.0)

Reserved, RES0.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode.

- | | |
|---|---|
| 0 | Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are not trapped to Hyp mode. |
| 1 | Non-secure EL0 and EL1 System register accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode. |

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to 0.

TDOSA, bit [10]

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode.

- | | |
|---|---|
| 0 | Non-secure EL1 System register accesses to the powerdown debug registers are not trapped to Hyp mode. |
| 1 | Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode. |

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, DBGOSDLR, and the DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Note

These registers are not accessible at PL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to 0.

TDA, bit [9]

Trap debug access. Traps Non-secure EL0 and EL1 System register accesses to those debug System registers in the (coproc==1110) encoding space that are not trapped by either of the following:

- HDCR.TDRA.
- HDCR.TDOSA.

- | | |
|---|---|
| 0 | Has no effect on System register accesses to the debug registers. |
| 1 | Non-secure EL0 or EL1 System register accesses to the debug registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA, are trapped to Hyp mode. |

HDCR.TDA does not trap accesses to the DBGDTRRXint or DBGDTRTXint when the PE is in Debug state.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to 0.

TDE, bit [8]

Trap Debug exceptions. The possible values of this bit are:

- | | |
|---|---|
| 0 | This control has no effect on the routing of debug exceptions, and has no effect on Non-secure accesses to debug registers. |
|---|---|

- 1 In Non-secure state:
- Debug exceptions generated at EL1 or EL0 are routed to EL2.
 - All accesses to Debug registers that would not be UNDEFINED if the value of this field was 0 are trapped to EL2.

When $\text{HCR.TGE} == 1$, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

When this register has an architecturally-defined reset value, this field resets to 0.

HPME, bit [7]

Hypervisor Performance Monitors Counters Enable. The possible values of this bit are:

- 0 Hyp mode Performance Monitors counters disabled.
- 1 Hyp mode Performance Monitors counters enabled.

When the value of this bit is 1, the Performance Monitors counters that are reserved for use from Hyp mode or Secure state are enabled. For more information see the description of the HPMN field.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

TPM, bit [6]

Trap Performance Monitors accesses. Traps Non-secure EL0 and EL1 accesses to all Performance Monitors registers to Hyp mode.

- 0 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are not trapped to Hyp mode.
- 1 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

————— Note —————

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

TPMCR, bit [5]

Trap **PMCR** accesses. Traps Non-secure EL0 and EL1 accesses to the **PMCR** to Hyp mode.

- 0 Non-secure EL0 and EL1 accesses to the **PMCR** are not trapped to Hyp mode.
- 1 Non-secure EL0 and EL1 accesses to the **PMCR** are trapped to Hyp mode.

————— Note —————

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

HPMN, bits [4:0]

Defines the number of Performance Monitors counters that are accessible from Non-secure EL1 modes, and from Non-secure EL0 modes if unprivileged access is enabled.

If the Performance Monitors Extension is not implemented, this field is RES0.

In Non-secure state, HPMN divides the Performance Monitors counters as follows. If software is accessing Performance Monitors counter n then, in Non-secure state:

- If n is in the range $0 \leq n < \text{HPMN}$, the counter is accessible from EL1 and EL2, and from EL0 if unprivileged access to the counters is enabled. **PMCR.E** enables the operation of counters in this range.
- If n is in the range $\text{HPMN} \leq n < \text{PMCR.N}$, the counter is accessible only from EL2 and from Secure state. **HDCR.HPME** enables the operation of counters in this range.

If this field is set to 0, or to a value larger than **PMCR.N**, then the following CONstrained UNPREDICTABLE behavior applies:

- The value returned by a direct read of **HDCR.HPMN** is UNKNOWN.
- Either:
 - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if **HDCR.HPMN** is set to an UNKNOWN non-zero value less than **PMCR.N**.
 - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to the value of **PMCR.N**.

Accessing the HDCR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 4, <Rt>, c1, c1, 1	100	001	0001	1111	0001

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 4, <Rt>, c1, c1, 1	x	x	0	-	-	n/a	-
p15, 4, <Rt>, c1, c1, 1	x	0	1	-	-	RW	RW
p15, 4, <Rt>, c1, c1, 1	x	1	1	-	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0):

- If HSTR_EL2.T1==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If HSTR_EL2.T1==1, Non-secure accesses to this register from EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and SCR_EL3.NS == 1:

- If HSTR.T1==1, Non-secure accesses to this register from EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If MDCR_EL3.TDA==1, accesses to this register from PL2 are trapped to EL3 using AArch64.

RETIRED

C6.4 Performance Monitors registers

This section lists the ARMv8.1 Performance Monitors registers in AArch32 state, in alphabetic order.

RETIRED

C6.4.1 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which common architectural and common microarchitectural feature events in the range 0x4000 to 0x401F are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

Configurations

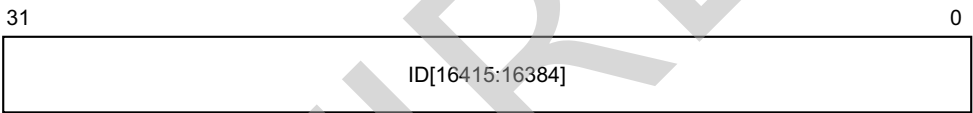
There is one instance of this register that is used in both Secure and Non-secure states.
AArch32 System register PMCEID2 is architecturally mapped to AArch64 System register [PMCEID0_EL0](#)[63:32].
AArch32 System register PMCEID2 is architecturally mapped to External register [PMCEID2](#)[63:32].

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

The PMCEID2 bit assignments are:



ID[16415:16384], bits [31:0]

PMCEID2[31:0] maps to common events 0x4000 to 0x401F. For a list of event numbers and descriptions, see [Events, event numbers, and mnemonics on page B12-557](#).
For each bit:
0 The common event is not implemented.
1 The common event is implemented.
Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.
Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Accessing the PMCEID2

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c9, c14, 4	000	100	1001	1111	1110

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c9, c14, 4	x	x	0	RO	RO	n/a	RO
p15, 0, <Rt>, c9, c14, 4	x	0	1	RO	RO	RO	RO
p15, 0, <Rt>, c9, c14, 4	x	1	1	RO	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR.EN==0, read accesses to this register from PL0 are trapped to Undefined mode.
- If PMUSERENR_EL0.EN==0, read accesses to this register from PL0 are trapped to EL1.

C6.4.2 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which common architectural and common microarchitectural feature events in the range 0x4020 to 0x403F are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

Configurations

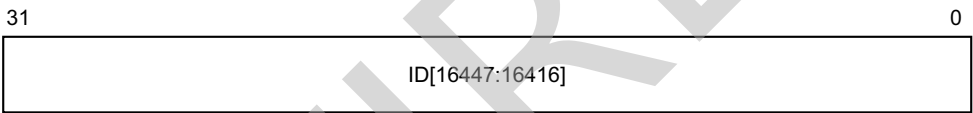
There is one instance of this register that is used in both Secure and Non-secure states.
AArch32 System register PMCEID3 is architecturally mapped to AArch64 System register [PMCEID1_EL0](#)[63:32].
AArch32 System register PMCEID3 is architecturally mapped to External register [PMCEID3](#)[63:32].

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

The PMCEID3 bit assignments are:



ID[16447:16416], bits [31:0]

PMCEID3[31:0] maps to common events 0x4020 to 0x403F. For a list of event numbers and descriptions, see [Events, event numbers, and mnemonics on page B12-557](#).

For each bit:

- 0 The common event is not implemented.
- 1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.
Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Accessing the PMCEID3

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c9, c14, 5	000	101	1001	1111	1110

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c9, c14, 5	x	x	0	RO	RO	n/a	RO
p15, 0, <Rt>, c9, c14, 5	x	0	1	RO	RO	RO	RO
p15, 0, <Rt>, c9, c14, 5	x	1	1	RO	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR.EN==0, read accesses to this register from PL0 are trapped to Undefined mode.
- If PMUSERENR_EL0.EN==0, read accesses to this register from PL0 are trapped to EL1.

C6.4.3 PMCR, Performance Monitors Control Register

The PMCR characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCR is architecturally mapped to AArch64 System register [PMCR_EL0](#).

AArch32 System register PMCR[6:0] is architecturally mapped to External register PMCR_EL0[6:0].

This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset these apply only if the PE resets into an Exception level that is using AArch32. Otherwise, on a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

PMCR is a 32-bit register.

Field descriptions

The PMCR bit assignments are:

31	24	23	16	15	11	10	7	6	5	4	3	2	1	0				
IMP				IDCODE				N		RES0		LC	DP	X	D	C	P	E

IMP, bits [31:24]

Implementer code. This field is RO with an IMPLEMENTATION DEFINED value.

The implementer codes are allocated by ARM. Values have the same interpretation as bits [31:24] of the MIDR.

IDCODE, bits [23:16]

Identification code. This field is RO with an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

N, bits [15:11]

Number of event counters. A RO field that indicates the number counters implemented. A value of 0b00000 in this field indicates that only the Cycle Count Register PMCCNTR is implemented.

The value of this field is the number of event counters implemented. This value is in the range of 0b00000, in which case only the PMCCNTR is implemented, to 0b11111, which indicates that the PMCCNTR and 31 event counters are implemented.

In an implementation that includes EL2, reads of this field from Non-secure EL1 and Non-secure EL0 return the value of [HDCR.HPMN](#) if EL2 is using AArch32, or the value of [MDCR_EL2.HPMN](#) if EL2 is using AArch64.

Bits [10:7]

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines which PMCCNTR bit generates an overflow recorded by PMOVSr[31].

- 0 Cycle counter overflow on increment that changes PMCCNTR[31] from 1 to 0.
- 1 Cycle counter overflow on increment that changes PMCCNTR[63] from 1 to 0.

ARM deprecates use of PMCR.LC = 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

DP, bit [5]

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

- 0 PMCCNTR, if enabled, counts when event counting is prohibited.
- 1 PMCCNTR does not count when event counting is prohibited.

Event counting is prohibited when `ProfilingProhibited(IsSecure(), PSTATE.EL) == TRUE`.

When EL3 is not implemented, this field is RES0:

- In ARMv8.0.
- In ARMv8.1, only if EL2 is not implemented.

Otherwise this field is RW.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

X, bit [4]

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

- 0 Do not export events.
- 1 Export events where not prohibited.

This field enables the exporting of events over an event bus to another device, for example to an OPTIONAL trace macrocell. If the implementation does not include such an event bus then this field is RAZ/WI, otherwise it is an RW field.

In an implementation that includes an event bus, no events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

D, bit [3]

Clock divider. The possible values of this bit are:

- 0 When enabled, PMCCNTR counts every clock cycle.
- 1 When enabled, PMCCNTR counts once every 64 clock cycles.

This bit is RW.

If PMCR.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR.D = 1.

When this register has an architecturally-defined reset value, this field resets to 0.

C, bit [2]

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

- 0 No action.
- 1 Reset PMCCNTR to zero.

This bit is always RAZ.

Resetting PMCCNTR does not clear the PMCCNTR overflow bit to 0.

P, bit [1]

Event counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset all event counters accessible in the current EL, not including PMCCNTR, to zero.

This bit is always RAZ.

In Non-secure EL0 and EL1, if EL2 is implemented, a write of 1 to this bit does not reset event counters that [HDCR.HPMN](#) or [MDCR_EL2.HPMN](#) reserves for EL2 use.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

Resetting the event counters does not clear any overflow bits to 0.

E, bit [0]

Enable. The possible values of this bit are:

0 All counters that are accessible at Non-secure EL1, including PMCCNTR, are disabled.

1 All counters that are accessible at Non-secure EL1 are enabled by PMCNTENSET.

This bit is RW.

If EL2 is implemented, this bit does not affect the operation of event counters that [HDCR.HPMN](#) or [MDCR_EL2.HPMN](#) reserves for EL2 use.

When this register has an architecturally-defined reset value, this field resets to 0.

Accessing the PMCR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c9, c12, 0	000	000	1001	1111	1100

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c9, c12, 0	x	x	0	RW	RW	n/a	RW
p15, 0, <Rt>, c9, c12, 0	x	0	1	RW	RW	RW	RW
p15, 0, <Rt>, c9, c12, 0	x	1	1	RW	n/a	RW	RW

Traps and Enables

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page C6-698 for exceptions taken to AArch32 state and *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547 for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If `PMUSERENR.EN==0`, accesses to this register from PL0 are trapped to Undefined mode.
- If `PMUSERENR_EL0.EN==0`, accesses to this register from PL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and `(SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 0)`:

- If `HSTR_EL2.T9==1`, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and `SCR_EL3.NS == 1`:

- If `MDCR_EL2.TPM==1`, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.
- If `MDCR_EL2.TPMCR==1`, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.

When EL2 is implemented and is using AArch64 and `(SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0)`:

- If `HSTR_EL2.T9==1`, Non-secure write accesses to this register from EL0 and EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and `SCR_EL3.NS == 1`:

- If `HDCR.TPM==1`, Non-secure accesses to this register from EL0 and EL1 are trapped to Hyp mode.
- If `HDCR.TPMCR==1`, Non-secure accesses to this register from EL0 and EL1 are trapped to Hyp mode.
- If `HSTR.T9==1`, Non-secure accesses to this register from EL0 and EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If `MDCR_EL3.TPM==1`, accesses to this register from PL0, PL1, and PL2 are trapped to EL3.

C6.4.4 PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n> characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMEVTYPER<n> is architecturally mapped to AArch64 System register [PMEVTYPER<n>_EL0](#).

AArch32 System register PMEVTYPER<n> is architecturally mapped to External register PMEVTYPER<n>_EL0.

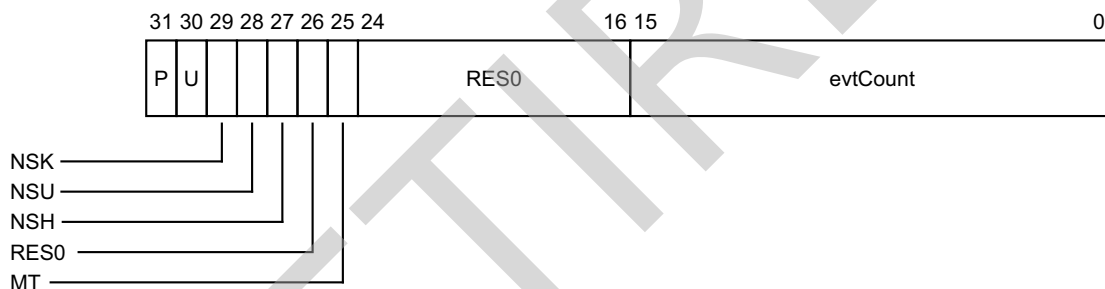
This register is in the Warm reset domain. On a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

Attributes

PMEVTYPER<n> is a 32-bit register.

Field descriptions

The PMEVTYPER<n> bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

- 0 Count events in EL1.
- 1 Do not count events in EL1.

U, bit [30]

User filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

- 0 Count events in EL0.
- 1 Do not count events in EL0.

NSK, bit [29]

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

NSU, bit [28]

Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

NSH, bit [27]

Non-secure EL2 (Hyp mode) filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

- | | |
|---|-----------------------------|
| 0 | Do not count events in EL2. |
| 1 | Count events in EL2. |

Bit [26]

Reserved, RES0.

MT, bit [25]

Multithreading. When the implementation is multi-threaded, the valid values for this bit are:

- | | |
|---|--|
| 0 | Count events only on controlling PE. |
| 1 | Count events from any PE with the same affinity at level 1 and above as this PE. |

When the implementation is not multi-threaded, this bit is RES0.

Note

- An implementation is described as multi-threaded when the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. That is, the performance of PEs at the lowest affinity level is highly interdependent. On such an implementation, the value of MPIDR_EL1.MT, when read at the highest implemented Exception level, is 1.
- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

Bits [24:16]

Reserved, RES0.

evtCount, bits [15:0] (In ARMv8.1)

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>.

Software must program this field with an event that is supported by the PE being programmed.

There are three ranges of event numbers:

- Event numbers in the range 0x000 to 0x03F are common architectural and microarchitectural events.
- Event numbers in the range 0x040 to 0x0BF are ARM recommended common architectural and microarchitectural events.
- Event numbers in the range 0x0C0 to 0x3FF are IMPLEMENTATION DEFINED events.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the event type:

- For the range 0x000 to 0x03F, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back from evtCount is UNKNOWN.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

Bits [15:10] (In ARMv8.0)

Reserved, RES0.

evtCount, bits [9:0] (In ARMv8.0)

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>.

Software must program this field with an event that is supported by the PE being programmed.

There are three ranges of event numbers:

- Event numbers in the range 0x000 to 0x03F are common architectural and microarchitectural events.
- Event numbers in the range 0x040 to 0x0BF are ARM recommended common architectural and microarchitectural events.
- Event numbers in the range 0x0C0 to 0x3FF are IMPLEMENTATION DEFINED events.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the event type:

- For the range 0x000 to 0x03F, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back from evtCount is UNKNOWN.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

Accessing the PMEVTYPER<n>

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c14, <CRm>, <opc2>	000	n<2:0>	1110	1111	11:n<4:3>

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c14, <CRm>, <opc2>	x	x	0	RW	RW	n/a	n/a
p15, 0, <Rt>, c14, <CRm>, <opc2>	x	0	1	RW	RW	RW	n/a
p15, 0, <Rt>, c14, <CRm>, <opc2>	x	1	1	RW	n/a	RW	n/a

This register is accessible at EL0 when PMUSERENR.EN is set to 1.

PMEVTYPER<n> can also be accessed by using PMXEVTYPER with PMSELR.SEL set to n.

If <n> is greater or equal to the number of accessible counters, reads and writes of PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- In Non-secure state, for an access from PL1 or a permitted access from PL0, if PMSELR.SEL, or PMSELR_EL0.SEL if EL1 is using AArch64, is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2. Accesses from PL0 are permitted when:
 - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
 - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

Note

In an implementation that includes EL2, in Non-secure state at EL0 and EL1:

- If EL2 is using AArch32, [HDCR](#).HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, [MDCR_EL2](#).HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters.

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

In both security states, and not dependent on other bits:

- If PMUSERENR.EN==0, accesses to this register from PL0 are trapped to Undefined mode.
- If PMUSERENR_EL0.EN==0, accesses to this register from PL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and SCR_EL3.NS == 1:

- If [MDCR_EL2](#).TPM==1, Non-secure accesses to this register from EL0 and EL1 are trapped to EL2.

When EL2 is implemented and is using AArch32 and $\text{SCR_EL3.NS} = 1$:

- If $\text{HDCR.TPM} = 1$, Non-secure accesses to this register from EL0 and EL1 are trapped to Hyp mode.

When EL3 is implemented and is using AArch64:

- If $\text{MDCR_EL3.TPM} = 1$, accesses to this register from PL0, PL1, and PL2 are trapped to EL3.

RETIRED

C6.5 Generic Timer registers

This section lists the ARMv8.1 Generic Timer registers in AArch32 state, in alphabetic order.

RETIRED

C6.5.1 CNTHV_CTL, Counter-timer Virtual Timer Control register (EL2)

The CNTHV_CTL characteristics are:

Purpose

Control register for the EL2 virtual timer.

Note

The EL2 virtual timer is only implemented in ARMv8.1, when EL2 is implemented and is using AArch64. It is only accessible at EL0 when [HCR_EL2](#).{E2H, TGE} is {1, 1}.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

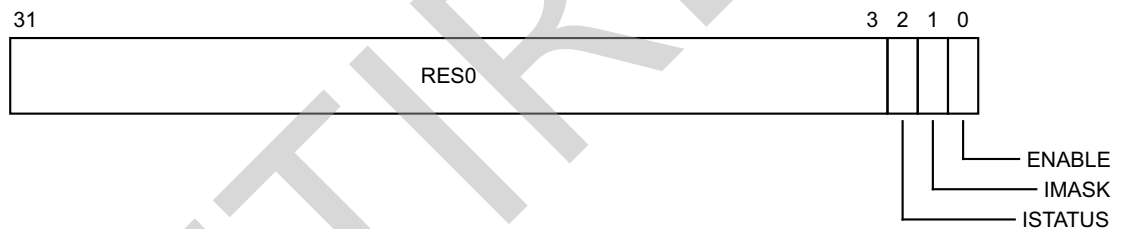
AArch32 System register CNTHV_CTL is architecturally mapped to AArch64 System register [CNTHV_CTL_EL2](#).

Attributes

CNTHV_CTL is a 32-bit register.

Field descriptions

The CNTHV_CTL bit assignments are:



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is asserted:

- 0 Timer condition is not asserted.
- 1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see “Operation of the CompareValue views of the timers” and “Operation of the TimerValue views of the timers” in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

This bit is read-only.

IMASK, bit [1]

Timer output signal mask bit. Permitted values are:

- 0 Timer output signal is not masked.
- 1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

ENABLE, bit [0]

Enables the timer. Permitted values are:

- 0 Timer disabled.
1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTHV_TVAL](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

Accessing the CNTHV_CTL

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c14, c3, 1	000	001	1110	1111	0011

This register is accessed using the encoding for CNTV_CTL.

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c14, c3, 1	x	x	0	CNTV_CTL	CNTV_CTL	n/a	CNTV_CTL
p15, 0, <Rt>, c14, c3, 1	0	0	1	CNTV_CTL	CNTV_CTL	CNTV_CTL	CNTV_CTL
p15, 0, <Rt>, c14, c3, 1	0	1	1	CNTV_CTL	n/a	CNTV_CTL	CNTV_CTL
p15, 0, <Rt>, c14, c3, 1	1	0	1	CNTV_CTL	CNTV_CTL	n/a	n/a
p15, 0, <Rt>, c14, c3, 1	1	1	1	RW	n/a	n/a	n/a

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHCTL_EL2.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

C6.5.2 CNTHV_CVAL, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL characteristics are:

Purpose

Holds the compare value for the EL2 virtual timer.

———— **Note** ————

The EL2 virtual timer is only implemented in ARMv8.1, when EL2 is implemented and is using AArch64. It is only accessible at EL0 when [HCR_EL2](#).{E2H, TGE} is {1, 1}.

Configurations

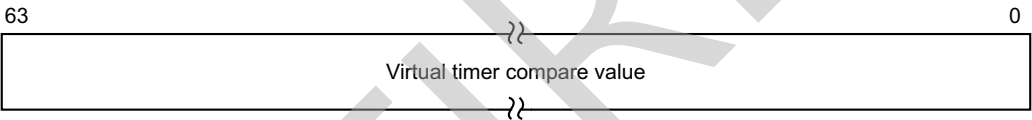
There is one instance of this register that is used in both Secure and Non-secure states.
AArch32 System register CNTHV_CVAL is architecturally mapped to AArch64 System register [CNTHV_CVAL_EL2](#).

Attributes

CNTHV_CVAL is a 64-bit register.

Field descriptions

The CNTHV_CVAL bit assignments are:



Bits [63:0]

Virtual timer compare value.

Accessing the CNTHV_CVAL

This register can be read using MRRC with the following syntax:

MRRC <syntax>

This register can be written using MCRR with the following syntax:

MCRR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	coproc	CRm
p15, 3, <Rt>, <Rt2>, c14	0011	1111	1110

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 3, <Rt>, <Rt2>, c14	x	x	0	CNTV_CVAL	CNTV_CVAL	n/a	CNTV_CVAL
p15, 3, <Rt>, <Rt2>, c14	0	0	1	CNTV_CVAL	CNTV_CVAL	CNTV_CVAL	CNTV_CVAL
p15, 3, <Rt>, <Rt2>, c14	0	1	1	CNTV_CVAL	n/a	CNTV_CVAL	CNTV_CVAL
p15, 3, <Rt>, <Rt2>, c14	1	0	1	CNTV_CVAL	CNTV_CVAL	n/a	n/a
p15, 3, <Rt>, <Rt2>, c14	1	1	1	RW	n/a	n/a	n/a

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHTCTL_EL2.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

C6.5.3 CNTHV_TVAL, Counter-timer Virtual Timer TimerValue register (EL2)

The CNTHV_TVAL characteristics are:

Purpose

Holds the timer value for the EL2 virtual timer.

———— **Note** ————

The EL2 virtual timer is only implemented in ARMv8.1, when EL2 is implemented and is using AArch64. It is only accessible at EL0 when [HCR_EL2](#).{E2H, TGE} is {1, 1}.

Configurations

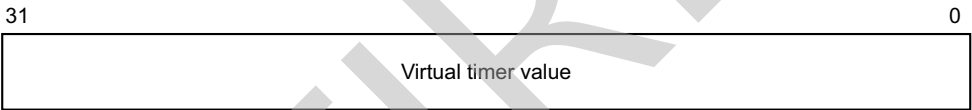
There is one instance of this register that is used in both Secure and Non-secure states.
AArch32 System register CNTHV_TVAL is architecturally mapped to AArch64 System register [CNTHV_TVAL_EL2](#).

Attributes

CNTHV_TVAL is a 32-bit register.

Field descriptions

The CNTHV_TVAL bit assignments are:



Bits [31:0]

Virtual timer value.

Accessing the CNTHV_TVAL

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	opc2	CRn	coproc	CRm
p15, 0, <Rt>, c14, c3, 0	000	000	1110	1111	0011

This register is accessed using the encoding for CNTV_TVAL.

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 0, <Rt>, c14, c3, 0	x	x	0	CNTV_TVAL	CNTV_TVAL	n/a	CNTV_TVAL
p15, 0, <Rt>, c14, c3, 0	0	0	1	CNTV_TVAL	CNTV_TVAL	CNTV_TVAL	CNTV_TVAL
p15, 0, <Rt>, c14, c3, 0	0	1	1	CNTV_TVAL	n/a	CNTV_TVAL	CNTV_TVAL
p15, 0, <Rt>, c14, c3, 0	1	0	1	CNTV_TVAL	CNTV_TVAL	n/a	n/a
p15, 0, <Rt>, c14, c3, 0	1	1	1	RW	n/a	n/a	n/a

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If [CNTHTCTL_EL2.EL0VTEN](#)==0, Non-secure accesses to this register from EL0 are trapped to EL2.

C6.5.4 CNTVCT, Counter-timer Virtual Count register

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT minus the virtual offset visible in CNTVOFF.

Configurations

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTVCT is architecturally mapped to AArch64 System register CNTVCT_EL0.

The value of this register is the same as the value of CNTPCT in the following conditions:

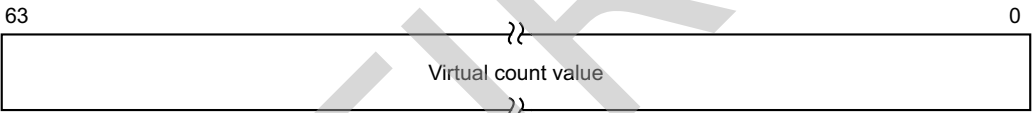
- When EL2 is not implemented.
- When EL2 is implemented and is using AArch64, HCR_EL2.{E2H, TGE} is {1, 1}, and this register is read from Non-secure EL0.

Attributes

CNTVCT is a 64-bit register.

Field descriptions

The CNTVCT bit assignments are:



Bits [63:0]

Virtual count value.

Accessing the CNTVCT

This register can be read using MRRC with the following syntax:

MRRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	coproc	CRm
p15, 1, <Rt>, <Rt2>, c14	0001	1111	1110

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 1, <Rt>, <Rt2>, c14	x	x	0	RO	RO	n/a	RO
p15, 1, <Rt>, <Rt2>, c14	x	0	1	RO	RO	RO	RO
p15, 1, <Rt>, <Rt2>, c14	x	1	1	RO	n/a	RO	RO

Traps and Enables

For a description of the prioritization of any generated exceptions, see [Synchronous exception prioritization for exceptions taken to AArch32 state on page C6-698](#) for exceptions taken to AArch32 state and [Synchronous exception prioritization for exceptions taken to AArch64 on page B12-547](#) for exceptions taken to AArch64 state. Subject to the prioritization rules:

When HCR_EL2.E2H == 0:

- If CNTKCTL.PL0VCTEN==0, read accesses to this register from PL0 are trapped to Undefined mode.
- If CNTKCTL_EL1.EL0VCTEN==0, read accesses to this register from PL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 0):

- If CNTKCTL.PL0VCTEN==0, Non-secure read accesses to this register from EL0 are trapped to Undefined mode.
- If CNTKCTL_EL1.EL0VCTEN==0, Non-secure read accesses to this register from EL0 are trapped to EL1.

When EL2 is implemented and is using AArch64 and (SCR_EL3.NS == 1) AND (HCR_EL2.E2H == 1) AND (HCR_EL2.TGE == 1):

- If CNTHCTL_EL2.EL0PCTEN==0, Non-secure read accesses to this register from EL0 are trapped to EL2.

C6.5.5 CNTVOFF, Counter-timer Virtual Offset register

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT and the virtual count value visible in CNTVCT.

Configurations

AArch32 System register CNTVOFF is architecturally mapped to AArch64 System register CNTVOFF_EL2.

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

———— Note ————

When EL2 is implemented and is using AArch64, if HCR_EL2.{E2H, TGE} is {1, 1}, the virtual counter uses a fixed virtual offset of zero when CNTVCT is read from Non-secure EL0.

When EL2 is implemented and can use AArch32, on a reset into an Exception level that is using AArch32 this register resets to an IMPLEMENTATION DEFINED value that might be UNKNOWN.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

The CNTVOFF bit assignments are:



Bits [63:0]

Virtual offset.

Accessing the CNTVOFF

This register can be read using MRRC with the following syntax:

MRRC <syntax>

This register can be written using MCRR with the following syntax:

MCRR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<syntax>	opc1	coproc	CRm
p15, 4, <Rt>, <Rt2>, c14	0100	1111	1110

Accessibility

The register is accessible in software as follows:

<syntax>	Control			Accessibility			
	E2H	TGE	NS	EL0	EL1	EL2	EL3
p15, 4, <Rt>, <Rt2>, c14	x	x	0	-	-	n/a	-
p15, 4, <Rt>, <Rt2>, c14	x	0	1	-	-	RW	RW
p15, 4, <Rt>, <Rt2>, c14	x	1	1	-	n/a	RW	RW

RETIRED

C6.6 ARMv8.0 sections relating to these registers

The following sections of the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* are included in this supplement to complement the register descriptions.

C6.6.1 Synchronous exception prioritization for exceptions taken to AArch32 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. This section describes the prioritization of such exceptions when they are taken to an Exception level that is using AArch32.

Note

- An exception that is taken to an Exception level that is using AArch32 must have been taken from an Exception level that is using AArch32.
 - The priority numbering in this list only shows the relative priorities of exceptions taken to an Exception level that is using AArch32. This numbering has no global significance and, for example, does not correlate with the equivalent AArch32 list in *Synchronous exception prioritization for exceptions taken to AArch64* on page B12-547.
-

For an exception that is taken to an Exception level that is using AArch32, exceptions are prioritized as follows, where 1 is the highest priority.

1. Misaligned PC exceptions. A Misaligned PC exception can only be taken to an Exception level that is using AArch32 as a result of:
 - The CONSTRAINED UNPREDICTABLE handling of a branch to an unaligned address.
 - Exiting from Debug state to AArch32 specifying an unaligned PC value.

A Misaligned PC exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception.

2. Prefetch Abort exceptions.
3. Breakpoint exceptions or Address Matching Vector Catchexceptions.

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in this section. This means that it does not have its own entry in this list.

4. Illegal Execution state exceptions.
5. Exceptions taken from EL1 to EL2 because of one of the following configuration settings:
 - HSTR.Tn.
 - HCR.TIDCP.
6. Undefined Instruction exceptions that occur as a result of one or more of the following:
 - An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
 - An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
 - Debug state execution of an instruction encoding that is unallocated in Debug state.
 - Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.
 - Execution of an HVC instruction, when HVC instructions are disabled by SCR.HCE or HCR.HCD.
 - Execution of an HLT instruction when HLT instructions are disabled by EDSCR.HDE.
 - In Debug state:
 - Execution of a DCPS1 instruction in Non-secure EL0 when HCR.TGE is 1.

- Execution of a DCP52 instruction in EL1 or EL0 when SCR.NS is 0 or when EL2 is not implemented.
 - Execution of a DCP53 instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
 - When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.
 - Execution of an instruction that is UNDEFINED as a result of any of:
 - Being in an IT block when SCTL.R.ITD is 1, or when HSCTL.R.ITD is 1.
 - Executing a SETEND instruction when SCTL.R.SED is 1, or when HSCTL.R.SED is 1.
 - Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTL.R.CP15BEN is 0, or when HSCTL.R.CP15BEN is 0.
 - Execution of an instruction that is UNDEFINED because at least one of FPSCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported.
7. Exceptions taken to EL1, or taken to EL2 because the value of HCR.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 1-6.
 8. Exceptions taken from EL0 to EL2 because of one of the following configuration settings:
 - HSTR.Tn.
 - HCR.TIDCP.
 9. Exceptions taken to EL2 because of configuration settings in the HCPTR.
 10. Exceptions taken to EL2 because of one of the following configuration settings:
 - Any setting in HCR, other than the TIDCP bit.
 - Any setting in CNTHCTL.
 - Any setting in HDCR.
 11. Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 1-10.
 12. Exceptions caused by the SMC instruction being UNDEFINED because the value of SCR.SCD is 1.
 13. Exceptions caused by the execution of an Exception generating instruction, SVC, HVC, SMC, or BKPT.
 14. Exceptions taken to EL3 because of configuration settings in the SDCR. These might be taken from EL0, EL1, or EL2.
 15. Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 1-14.
 16. Trapped floating-point exceptions, if supported.
 17. Data Abort exceptions other than a Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk. That is, any Data Abort exception that is not covered by item 19. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 19.
 18. Watchpoint exceptions.
 19. Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 17.

For items 17-19, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

———— **Note** ————

Exceptions generated by a translation table walk are reported and prioritized as either a Prefetch Abort exception, priority 2 in this list, or a Data Abort exception, priority 17 in this list. See also *AArch32 prioritization of synchronous aborts from a single stage of address translation in Chapter G4 of the ARM ARM*.

C6.6.2 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch64 definitions, as given in [Reserved values in System and memory-mapped registers and translation table entries](#) on page B12-558.

C6.6.3 Using the BAS field in Address Match breakpoints

See Using the BAS field in Address Match breakpoints under the Breakpoint exceptions section in Chapter G2 AArch32 Self-hosted Debug of the ARMv8 ARM.

C6.6.4 Usage constraints

See Usage constraints under the Breakpoint exceptions section in Chapter G2 AArch32 Self-hosted Debug of the ARMv8 ARM.

Part D

ARMv8.1 Changes to External Debug

RETIRED

Chapter D1

PC Sample-based Profiling

This chapter describes the changes to the PC Sample-based Profiling Extension in ARMv8.1. It contains the following section:

- [Changes to PC Sample-based profiling on page D1-704.](#)

D1.1 Changes to PC Sample-based profiling

A control bit, SC2, is added to EDSCR to control whether PC Sample-based profiling samples CONTEXTIDR_EL2 or VTTBR_EL2.VMID value.

When the value of EDSCR.SC2 is 0:

- EDPCSRlo is unchanged.
- [EDVIDSR](#).VMID is extended to EDVIDSR[15:0] to support sampling of the 16-bit VMID when the value of VTCR_EL2.VS is 1.

When EDSCR.SC2 is 1:

- Reading EDPCSRlo has a side-effect of updating EDCIDSR, [EDVIDSR](#), and EDPCSRhi:
 - The value written to EDCIDSR is the same as when SC2 is set to 0.
 - If the sampled PC is from Non-secure state and EL2 is using AArch64, [EDVIDSR](#) is written with the value of [CONTEXTIDR_EL2](#) associated with the most recent EDPCR sample. Otherwise, [EDVIDSR](#) becomes UNKNOWN.
 - EDPCSRhi[23:0] records bits[53:32] of the sampled PC and EDPCSRhi[31:29] record the SCR.NS bit state and the current Exception level.
- When any of the following applies, the PE behaves as if the value of [EDSCR](#).SC2 is 0 for all purposes other than a direct read of [EDSCR](#):
 - The PE is in Debug state.
 - The PE is in Reset state.
 - Sample-based profiling is prohibited.
 - No instruction has been retired since the PE left Reset state.

D1.1.1 Identification mechanism

The [EDDEVARCH](#).ARCHID[15:12] field identifies the support for the v8.1 changes to PC Sample-based profiling.

D1.1.2 See also

In this supplement

- [EDDEVARCH](#).ARCHID[15:12].
- [EDPCSR](#).
- [EDSCR](#).SC2.
- [EDVIDSR](#).
- [Appendix F1 Notes on Using Debug and Performance Monitors](#).

In the ARM Architecture Reference Manual

The PC Sample-based Profiling Extension.

Chapter D2

External Debug Register Descriptions

This chapter describes the External debug registers that are added or or affected by ARMv8.1, or by the changes to the Performance Monitors Extension introduced with ARMv8.1. It contains the following sections:

- [General information about External debug register descriptions on page D2-706.](#)
- [Debug registers on page D2-707.](#)
- [Performance Monitors registers on page D2-733](#)

D2.1 General information about External debug register descriptions

This chapter provides full descriptions of all of the registers that are accessible through the external debug interface and are affected by the introduction of ARMv8.1, or by the changes to the Performance Monitors Extension introduced with ARMv8.

The registers descriptions in this chapter do not highlight where ARMv8.1 has changed the register field descriptions. However:

- The field descriptions indicate any differences in behavior between ARMv8.0 and ARMv8.1.
- The descriptions of the features of ARMv8.1 elsewhere in this manual indicate where ARMv8.1 has introduced new register fields, or significantly changed the effect of a register field.

Note

The structure of the descriptions of memory-mapped registers, including all register that can be accessed through the external debug interface, is unchanged between ARMv8.0 and ARMv8.1. That is, the restructuring of System register descriptions described in [General information about AArch64 System registers on page B12-232](#) and [General information about AArch32 System registers on page C6-590](#) does not apply to these memory-mapped registers.

D2.2 Debug registers

This section provides full descriptions of all of the debug registers that are accessible through the external debug interface and are affected by the introduction of ARMv8.1. See [General information about External debug register descriptions on page D2-706](#) for more information about the descriptions of these registers.

RETIRED

D2.2.1 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	EDAD	SLK	Default
Error	Error	Error	Error	RO	RW

Configurations

External register DBGBCR<n>_EL1 is architecturally mapped to AArch64 System register DBGBCR<n>_EL1.

External register DBGBCR<n>_EL1 is architecturally mapped to AArch32 System register DBGBCR<n>.

DBGBCR<n>_EL1 is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External debug reset.

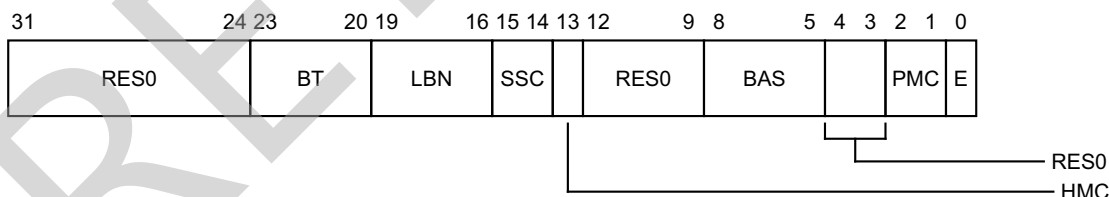
If breakpoint n is not implemented then this register is unallocated.

Attributes

DBGBCR<n>_EL1 is a 32-bit register.

Field descriptions

The DBGBCR<n>_EL1 bit assignments are:



When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

0000	Unlinked address match.
0001	Linked address match.
0010	Unlinked Context ID match.
0011	Linked Context ID match.
0100	Unlinked instruction address mismatch.
0101	Linked instruction address mismatch.

0110	Unlinked CONTEXTIDR_EL1 match (ARMv8.1).
0111	Linked CONTEXTIDR_EL1 match (ARMv8.1).
1000	Unlinked VMID match.
1001	Linked VMID match.
1010	Unlinked VMID and Context ID match.
1011	Linked VMID and Context ID match.
1100	Unlinked CONTEXTIDR_EL2 match (ARMv8.1).
1101	Linked CONTEXTIDR_EL2 match (ARMv8.1).
1110	Unlinked Full Context ID match (ARMv8.1).
1111	Linked Full Context ID match (ARMv8.1).

The field breaks down as follows:

- BT[3:1]: Base type.

000	Match address. DBGBVR<n>_EL1 is the address of an instruction.
001	Match Context ID. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1 in ARMv8.0, and in ARMv8.1 when not in a Host OS or a Host Application. In ARMv8.1, when in a Host OS or Host Application, the Context ID is compared against CONTEXTIDR_EL1 .
010	Mismatch address. DBGBVR<n>_EL1 is the address of an instruction to be stepped.
011	Match CONTEXTIDR_EL1 . DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1 .
100	Match VMID. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID .
101	Match VMID and Context ID. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1 , and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID .
110	Match CONTEXTIDR_EL2 . DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2 .
111	Match Full Context ID. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1 , and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2 .
- BT[0]: Enable linking.

All other values are reserved. Constraints on breakpoint programming mean other values are reserved under some conditions. For more information, including the effect of programming this field to a reserved value, see [Reserved DBGBCR<n>_EL1.BT values on page B8-70](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of [DBGBCR<n>_EL1.E](#) is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint *n* is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see "Reserved DBGBCR<*n*>_EL1.{SSC,HMC,PMC} values" in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint *n* is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see [DBGBCR<*n*>_EL1.SSC](#) description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state. In an AArch64-only implementation, this field is reserved, RES1.

The permitted values depend on the breakpoint type.

For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0011	DBGBVR<<i>n</i>>_EL1	Use for T32 instructions.
1100	DBGBVR<<i>n</i>>_EL1+2	Use for T32 instructions.
1111	DBGBVR<<i>n</i>>_EL1	Use for A64 and A32 instructions.

All other values are reserved.

For more information, see [Using the BAS field in Address Match breakpoints on page C6-700](#).

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Step instruction at	Constraint for debuggers
0000	-	Use for a match anywhere breakpoint.
0011	DBGBVR<<i>n</i>>_EL1	Use for stepping T32 instructions.
1100	DBGBVR<<i>n</i>>_EL1+2	Use for stepping T32 instructions.
1111	DBGBVR<<i>n</i>>_EL1	Use for stepping A64 and A32 instructions.

For more information, see [Using the BAS field in Address Match breakpoints on page C6-700](#).

For Context matching breakpoints, this field is RES1 and ignored.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the [DBGBCR<n>_EL1](#).SSC description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

E, bit [0]

Enable breakpoint [DBGBVR<n>_EL1](#). Possible values are:

0 Breakpoint disabled.

1 Breakpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the DBGBCR<n>_EL1:

DBGBCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset
Debug	$0x408 + 16n$

D2.2.2 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register [DBGBCR<n>_EL1](#).

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	EDAD	SLK	Default
Error	Error	Error	Error	RO	RW

Configurations

External register DBGBVR<n>_EL1 is architecturally mapped to AArch64 System register [DBGBVR<n>_EL1](#).

External register DBGBVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register [DBGBVR<n>](#).

External register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register [DBGXVR<n>](#).

DBGBVR<n>_EL1 is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External debug reset.

If breakpoint n is not implemented then this register is unallocated.

Attributes

How this register is interpreted depends on the value of [DBGBCR<n>_EL1.BT](#).

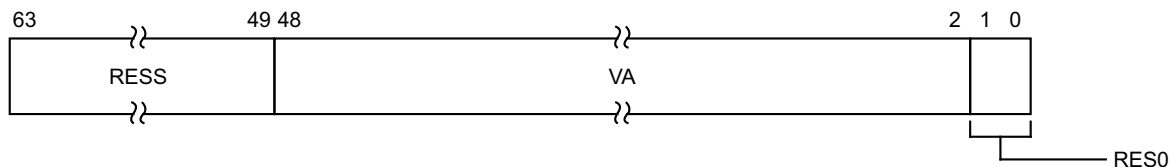
- When [DBGBCR<n>_EL1.BT](#) is 0b0x0x, this register holds a virtual address.
- When [DBGBCR<n>_EL1.BT](#) is 0b001x, 0b011x, or 0b110x, this register holds a Context ID.
- When [DBGBCR<n>_EL1.BT](#) is 0b100x, this register holds a VMID.
- When [DBGBCR<n>_EL1.BT](#) is 0b101x, this register holds a VMID and a Context ID.
- When [DBGBCR<n>_EL1.BT](#) is 0b111x, this register holds two Context ID values.

For other values of [DBGBCR<n>_EL1.BT](#), this register is RES0.

Field descriptions

The DBGBVR<n>_EL1 bit assignments are:

When [DBGBCR<n>_EL1.BT](#)==0b0x0x:



RESS, bits [63:49]

Reserved, Sign extended. Software must treat this field as RES0 if bit[48] is 0 or RES0, and as RES1 if bit[48] is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of bit [48], meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

VA, bits [48:2]

If the address is being matched in an AArch64 stage 1 translation regime, this field contains bits[48:2] of the address for comparison.

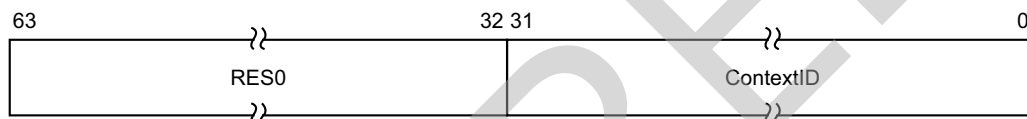
If the address is being matched in an AArch32 stage 1 translation regime, the first 16 bits of this field are RES0, and the rest of the field contains bits[31:2] of the address for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [1:0]

Reserved, RES0.

When $DBGBCR<n>_{EL1.BT}=0b001x$:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR and [CONTEXTIDR_EL1](#) in the following cases:

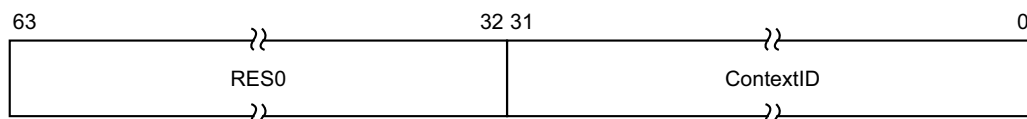
- The PE is in Secure state.
- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.
- In ARMv8.1, when EL2 is using AArch64, [HCR_EL2.E2H](#) is 0 and the PE is in Non-secure EL0, EL1 or EL2.
- In ARMv8.1, when EL2 is using AArch64, [HCR_EL2](#).{E2H, TGE} is {1, 0} and the PE is in Non-secure EL0 or EL1.

In ARMv8.1, when EL2 is using AArch64 and [HCR_EL2.E2H](#) is 1, the value is compared against [CONTEXTIDR_EL2](#) in the following cases:

- The PE is executing at EL2.
- [HCR_EL2.TGE](#) is 1 and the PE is in Non-secure EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>_{EL1.BT}=0b011x$:



Bits [63:32]

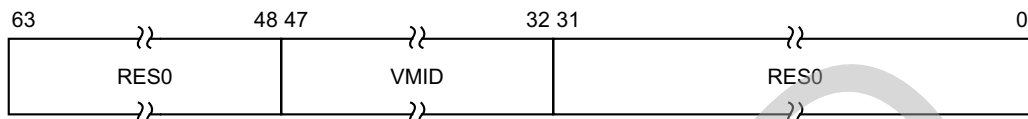
Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL1](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>_{EL1}.BT == 0b100x$ and EL2 implemented:



Bits [63:48]

Reserved, RES0.

VMID, bits [47:32] (In ARMv8.1)

VMID value for comparison.

The VMID is 8 bits in the following cases.

- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.

In ARMv8.1 when EL2 is using AArch64, it is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

The upper 8 bits of this field are RES0 if any of the following apply:

- The implementation has an 8 bit VMID.
- [VTCR_EL2.VS](#) is 0.
- EL2 is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [47:40] (In ARMv8.0)

Reserved, RES0.

VMID, bits [39:32] (In ARMv8.0)

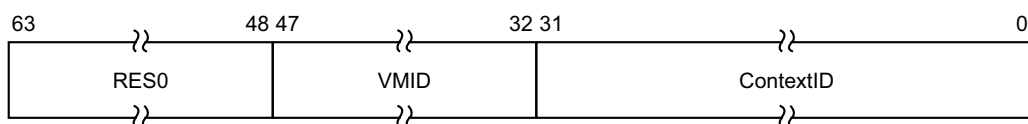
VMID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [31:0]

Reserved, RES0.

When $DBGBCR<n>_{EL1}.BT == 0b101x$ and EL2 implemented:



Bits [63:48]

Reserved, RES0.

VMID, bits [47:32] (In ARMv8.1)

VMID value for comparison.

The VMID is 8 bits in the following cases.

- In ARMv8.0.
- In ARMv8.1, when EL2 is using AArch32.

In ARMv8.1 when EL2 is using AArch64, it is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

The upper 8 bits of this field are RES0 if any of the following apply:

- The implementation has an 8 bit VMID.
- [VTCR_EL2.VS](#) is 0.
- EL2 is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [47:40] (In ARMv8.0)

Reserved, RES0.

VMID, bits [39:32] (In ARMv8.0)

VMID value for comparison.

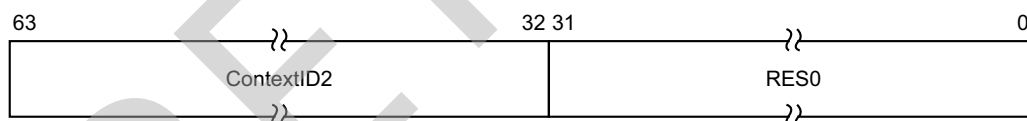
When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL1](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When $DBGBCR<n>_EL1.BT == 0b110x$ and EL2 implemented:



ContextID2, bits [63:32]

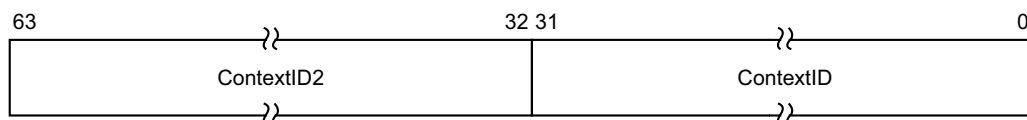
Context ID value for comparison against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [31:0]

Reserved, RES0.

When $DBGBCR<n>_EL1.BT == 0b111x$ and EL2 implemented:



ContextID2, bits [63:32]

Context ID value for comparison against [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR_EL1](#).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the DBGBVR<n>_EL1:

DBGBVR<n>_EL1[31:0] can be accessed through the external debug interface:

Component	Offset
Debug	$0x400 + 16n$

DBGBVR<n>_EL1[63:32] can be accessed through the external debug interface:

Component	Offset
Debug	$0x404 + 16n$

D2.2.3 EDCIDSR, External Debug Context ID Sample Register

The EDCIDSR characteristics are:

Purpose

Contains the sampled value of the Context ID, captured on reading the low half of [EDPCSR](#).

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	Default
Error	Error	Error	RO

Configurations

EDCIDSR is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External debug reset.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented.

Attributes

EDCIDSR is a 32-bit register.

Field descriptions

The EDCIDSR bit assignments are:



CONTEXTIDR, bits [31:0]

The sampled value of the Context ID, captured on reading the low half of [EDPCSR](#).

If EL1 is using AArch64, the Context ID is held in [CONTEXTIDR_EL1](#).

If EL1 is using AArch32, the Context ID is held in CONTEXTIDR. If EL3 is implemented and is using AArch32 then CONTEXTIDR is a Banked register, and EDCIDSR samples the current Banked copy of CONTEXTIDR.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the EDCIDSR:

EDCIDSR can be accessed through the external debug interface:

Component	Offset
Debug	0x0A4

D2.2.4 EDDEVARCH, External Debug Device Architecture register

The EDDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the external debug component.

Usage constraints

This register is accessible as follows:

Default

RO

Configurations

EDDEVARCH is in the Debug power domain.

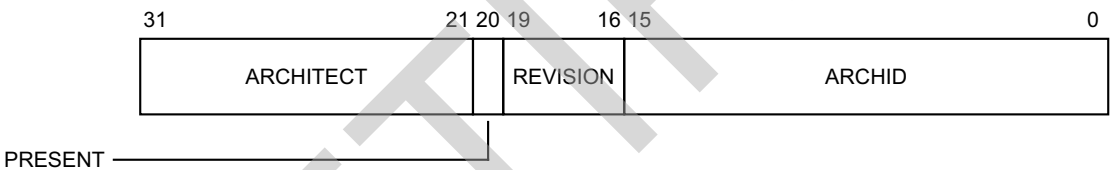
Implementation of this register is OPTIONAL.

Attributes

EDDEVARCH is a 32-bit register.

Field descriptions

The EDDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Defines the architecture of the component. For debug, this is ARM Limited.

Bits [31:28] are the JEP 106 continuation code, 0x4.

Bits [27:21] are the JEP 106 ID code, 0x3B.

PRESENT, bit [20]

When set to 1, indicates that the DEVARCH is present.

This field is 1 in ARMv8.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by ARM this is the minor revision.

For debug, the revision defined by ARMv8 is 0x0.

All other values are reserved.

ARCHID, bits [15:0]

Defines this part to be an ARMv8 debug component. For architectures defined by ARM this is further subdivided.

For debug:

- Bits [15:12] are the architecture version:
 - In ARMv8-A, this is 0x6.

- In ARMv8.1, this is 0x7.
 - Bits [11:0] are the architecture part number, 0xA15.
- This corresponds to the ARMv8 debug architecture version.

Accessing the EDDEVARCH:

EDDEVARCH can be accessed through the external debug interface:

Component	Offset
Debug	0xFBC

RETIRED

D2.2.5 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

Purpose

Provides top level information about the debug system in AArch64.

Usage constraints

This register is accessible as follows:

Default

RO

Note

Debuggers must use [EDDEVARCH](#) to determine the Debug architecture version.

Configurations

EDDFR is in the Debug power domain.

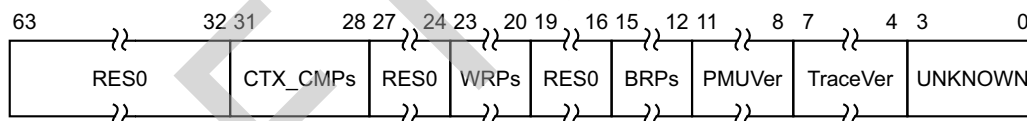
In an ARMv8-A implementation, this register gives information from the AArch64 register [ID_AA64DFR0_EL1](#).

Attributes

EDDFR is a 64-bit register.

Field descriptions

The EDDFR bit assignments are:



Bits [63:32]

Reserved, RES0.

CTX_CMPs, bits [31:28]

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

In an ARMv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of [ID_AA64DFR0_EL1](#).CTX_CMPs.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1. The value of 0b0000 is reserved.

In an ARMv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of [ID_AA64DFR0_EL1](#).WRPs.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1. The value of 0b0000 is reserved.

In an ARMv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of [ID_AA64DFR0_EL1](#).BRPs.

PMUVer, bits [11:8]

Performance Monitors extension version. Indicates whether System register interface to Performance Monitors extension is implemented. Defined values are:

- 0000 Performance Monitors extension System registers not implemented.
- 0001 Performance Monitors extension System registers implemented, PMUv3.
- 0100 Performance Monitors extension System registers implemented, PMUv3, with a 16-bit evtCount field, and if EL2 is implemented, the [MDCR_EL2](#).HPMD bit is meaningful.
- 1111 IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0001 and 1111.

In ARMv8.1 the permitted values are 0000, 0100 and 1111.

In an ARMv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of [ID_AA64DFR0_EL1](#).PMUVer.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace macrocell is implemented. Defined values are:

- 0000 Trace macrocell System registers not implemented.
- 0001 Trace macrocell System registers implemented.

All other values are reserved.

A value of 0000 only indicates that no System register interface to a trace macrocell is implemented. A trace macrocell might nevertheless be implemented without a System register interface.

In an ARMv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of [ID_AA64DFR0_EL1](#).TraceVer.

UNKNOWN, bits [3:0]

Reserved, UNKNOWN.

Accessing the EDDFR:

EDDFR[31:0] can be accessed through the external debug interface:

Component	Offset
Debug	0xD28

EDDFR[63:32] can be accessed through the external debug interface:

Component	Offset
Debug	0xD2C

D2.2.6 EDPCSR, External Debug Program Counter Sample Register

The EDPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	SLK	Default
Error	Error	Error	RO	RO

Configurations

EDPCSR is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External debug reset.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented.

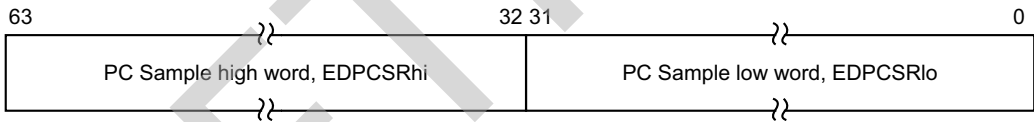
Attributes

In ARMv8.1, the format of this register differs depending on the value of [EDSCR.SC2](#).

Field descriptions

The EDPCSR bit assignments are:

When [EDSCR.SC2](#) == 0:



This format applies in all ARMv8.0 implementations.

Bits [63:32]

PC Sample high word, EDPCSRhi. If [EDVIDSR.HV](#) == 0 then this field is RAZ, otherwise bits [63:32] of the sampled instruction address value.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [31:0]

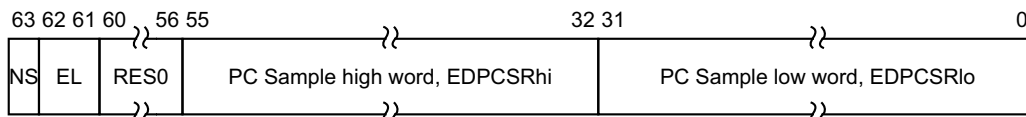
PC Sample low word, EDPCSRlo. Bits [31:0] of the sampled instruction address value. Reading EDPCSRlo has the side-effect of updating [EDCIDSR](#), [EDVIDSR](#), and EDPCSRhi. However:

- If the PE is in Debug state, or PC Sample-based profiling is prohibited, EDPCSRlo reads as 0xFFFFFFFF and [EDCIDSR](#), [EDVIDSR](#), and EDPCSRhi become UNKNOWN.
- If the PE is in Reset state, the sampled value is unknown and [EDCIDSR](#), [EDVIDSR](#) and EDPCSRhi become UNKNOWN.
- If no instruction has been retired since the PE left Reset state, Debug state, or a state where Non-invasive debug is not permitted, the sampled value is UNKNOWN and [EDCIDSR](#), [EDVIDSR](#), and EDPCSRhi become UNKNOWN.

- For a read of EDPCRlo from the memory-mapped interface, if EDLSR.SLK == 1, meaning the Software Lock is locked, then the access has no side-effects. That is, EDCIDSR, EDVIDSR, and EDPCRhi are unchanged.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When `EDSCR.SC2 == 1`:

**NS, bit [63]**

Non-secure state sample. Indicates the Security state associated with the most recent **EDPCSR** sample.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level associated with the most recent [EDPCSR](#) sample.

- | | |
|----|---------------------|
| 00 | Sample is from EL0. |
| 01 | Sample is from EL1. |
| 10 | Sample is from EL2. |
| 11 | Sample is from EL3. |

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [60:56]

Reserved. RES0.

Bits [55:32]

PC Sample high word, EDPCRhi. Bits [55:32] of the sampled instruction address value.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [31:0]

PC Sample low word, EDPCSRlo. Bits [31:0] of the sampled instruction address value. Reading EDPCSRlo has the side-effect of updating [EDCIDSr](#), [EDVIDSR](#), and EDPCSRhi. However:

- If the PE is in Debug state, or PC Sample-based profiling is prohibited, EDPCSRlo reads as 0xFFFFFFFF and [EDCIDSR](#), [EDVIDSR](#), and EDPCSRhi become UNKNOWN.
- If the PE is in Reset state, the sampled value is unknown and [EDCIDSR](#), [EDVIDSR](#) and EDPCSRhi become UNKNOWN.
- If no instruction has been retired since the PE left Reset state, Debug state, or a state where Non-invasive debug is not permitted, the sampled value is UNKNOWN and [EDCIDSR](#), [EDVIDSR](#), and EDPCSRhi become UNKNOWN.
- For a read of EDPCSRlo from the memory-mapped interface, if EDLSR.SLK == 1, meaning the Software Lock is locked, then the access has no side-effects. That is, [EDCIDSR](#), [EDVIDSR](#), and EDPCSRhi are unchanged.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the EDPCSR:

EDPCSR[31:0] can be accessed through the external debug interface:

Component	Offset
Debug	0x0A0

EDPCSR[63:32] can be accessed through the external debug interface:

Component	Offset
Debug	0x0AC

RETIRED

D2.2.7 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	SLK	Default
Error	Error	Error	RO	RW

Configurations

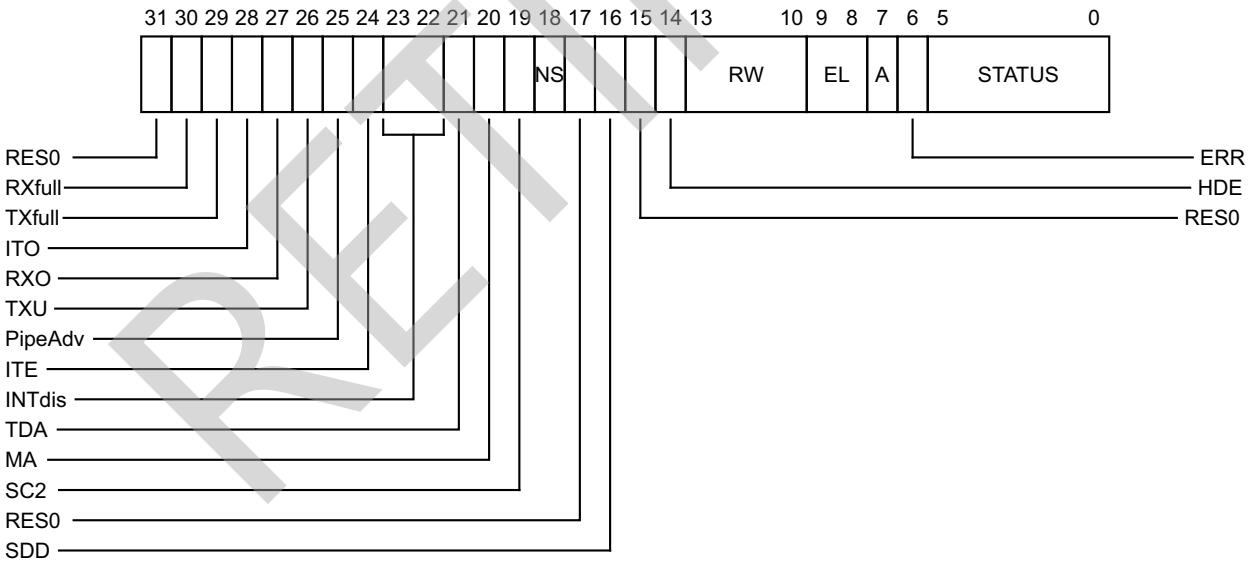
EDSCR is in the Core power domain. Some or all RW fields of this register have defined reset values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External debug reset.

Attributes

EDSCR is a 32-bit register.

Field descriptions

The EDSCR bit assignments are:



Bit [31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to 0.

TXfull, bit [29]

DTRTX full. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to 0.

ITO, bit [28]

ITR overrun. This bit is RO.

If the PE is in Non-debug state, this bit is UNKNOWN. ITO is set to 0 on entry to Debug state.

R XO, bit [27]

DTRRX overrun. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to 0.

TXU, bit [26]

DTRTX underrun. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to 0.

PipeAdv, bit [25]

Pipeline advance. This bit is RO. Set to 1 every time the PE pipeline retires one or more instructions. Cleared to 0 by a write to EDSCR.CSPA.

The architecture does not define precisely when this bit is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing.

ITE, bit [24]

ITR empty. This bit is RO.

If the PE is in Non-debug state, this bit is UNKNOWN. It is always valid in Debug state.

INTdis, bits [23:22]

Interrupt disable. Disables taking interrupts (including virtual interrupts and System Error interrupts) in Non-Debug state.

If external invasive debug is disabled, the value of this field is ignored.

If external invasive debug is enabled, the possible values of this field are:

- | | |
|----|--|
| 00 | Do not disable interrupts. |
| 01 | Disable interrupts taken to Non-secure EL1. |
| 10 | Disable interrupts taken only to Non-secure EL1 and Non-secure EL2. If external secure invasive debug is enabled, also disable interrupts taken to Secure EL1. |
| 11 | Disable interrupts taken only to Non-secure EL1 and Non-secure EL2. If external secure invasive debug is enabled, also disable all other interrupts. |

The value of INTdis does not affect whether an interrupt is a WFI wake-up event, but can mask an interrupt as a WFE wake-up event.

If EL3 and EL2 are not implemented, the values 0b01 and 0b10 are reserved. If programmed with a reserved value the PE behaves as if INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.

When this register has an architecturally-defined reset value, this field resets to 0.

TDA, bit [21]

Traps accesses to the following Debug System registers:

- AArch64: [DBGBCR<n>_EL1](#), [DBGBVR<n>_EL1](#), [DBGWCR<n>_EL1](#), [DBGWVR<n>_EL1](#).
- AArch32: [DBGBCR<n>](#), [DBGBVR<n>](#), [DBGXVR<n>](#), [DBGWCR<n>](#), [DBGWVR<n>](#).

The possible values of this field are:

- | | |
|---|--|
| 0 | Accesses to Debug System registers do not generate a Software Access debug event. |
| 1 | Accesses to Debug System registers generate a Software Access debug event, if OSLSR.OSLK is 0 and if halting is allowed. |

When this register has an architecturally-defined reset value, this field resets to 0.

MA, bit [20]

Memory access mode. Controls use of memory-access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

- 0 Normal access mode.
- 1 Memory access mode.

When this register has an architecturally-defined reset value, this field resets to 0.

SC2, bit [19] (In ARMv8.1)

Sample [CONTEXTIDR_EL2](#). Controls whether the Sample-based Profiling Extension samples [CONTEXTIDR_EL2](#) or [VTTBR_EL2.VMID](#).

- 0 Sample [VTTBR_EL2.VMID](#).
- 1 Sample [CONTEXTIDR_EL2](#).

When this register has an architecturally-defined reset value, this field resets to 0.

Bit [19] (In ARMv8.0)

Reserved, RES0.

NS, bit [18]

Non-secure status. Read-only. When in Debug state, gives the current Security state:

- 0 Secure state, `IsSecure() == TRUE`.
- 1 Non-secure state, `IsSecure() == FALSE`.

In Non-debug state, this bit is UNKNOWN.

Bit [17]

Reserved, RES0.

SDD, bit [16]

Secure debug disabled. This bit is RO.

On entry to Debug state:

- If entering in Secure state, SDD is set to 0.
- If entering in Non-secure state, SDD is set to the inverse of `ExternalSecureInvasiveDebugEnabled()`.

In Debug state, the value of the SDD bit does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

In Non-debug state:

- SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, a context synchronization operation is required to guarantee their effect.
- This bit is unaffected by the Security state of the PE.

If EL3 is not implemented and the implementation is Non-secure, this bit is RES1.

Bit [15]

Reserved, RES0.

HDE, bit [14]

Halting debug enable. The possible values of this field are:

- 0 Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
- 1 Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

When this register has an architecturally-defined reset value, this field resets to 0.

RW, bits [13:10]

Exception level Execution state status. Read-only. In Debug state, each bit gives the current Execution state of each EL:

RW	Meaning
1111	All Exception levels are using AArch64.
1110	EL0 is using AArch32. All other Exception levels are using AArch64.
110x	EL0 and EL1 are using AArch32. All other Exception levels are using AArch64. Never seen if EL2 is not implemented in the current Security state.
10xx	EL0, EL1, and, if implemented in the current Security state, EL2 are using AArch32. All other Exception levels are using AArch64.
0xxx	All Exception levels are using AArch32.

However:

- The value of 1110 is only permitted at EL0.
- The values 110x are not permitted if either:
 - EL2 is not implemented.
 - EL3 is implemented and SCR_EL3.NS/SCR.NS == 0.
- The values 10xx are not permitted if EL3 is not implemented.

In Non-debug state, this field is RAO.

EL, bits [9:8]

Exception level. Read-only. In Debug state, this gives the current EL of the PE.

In Non-debug state, this field is RAZ.

A, bit [7]

System Error interrupt pending. Read-only. In Debug state, indicates whether a SError interrupt is pending:

- If [HCR_EL2](#).{AMO, TGE} = {1, 0} and in Non-secure EL0 or EL1, a virtual SError interrupt.
- Otherwise, a physical SError interrupt.

0 No SError interrupt pending.

1 SError interrupt pending.

A debugger can read EDSCR to check whether an SError interrupt is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

UNKNOWN in Non-debug state.

ERR, bit [6]

Cumulative error flag. This field is RO. It is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When this register has an architecturally-defined reset value, this field resets to 0.

STATUS, bits [5:0]

Debug status flags. This field is RO.

The possible values of this field are:

000010 PE is in Non-debug state.

000001 PE is restarting, exiting Debug state.

000111	Breakpoint.
010011	External debug request.
011011	Halting step, normal.
011111	Halting step, exclusive.
100011	OS Unlock Catch.
100111	Reset Catch.
101011	Watchpoint.
101111	HLT instruction.
110011	Software access to debug register.
110111	Exception Catch.
111011	Halting step, no syndrome.
All other values of STATUS are reserved.	

Accessing the EDSCR:

EDSCR can be accessed through the external debug interface:

Component	Offset
Debug	0x088

D2.2.8 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

Purpose

Contains sampled values captured on reading [EDPCSR](#).

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	Default
Error	Error	Error	RO

Configurations

EDVIDSR is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External debug reset.

Required only if the OPTIONAL PC Sample-based Profiling Extension is implemented and the implementation includes at least one of EL2 and EL3. In an implementation that includes the PC Sample-based Profiling Extension and has EL1 as the highest implemented exception level, EDVIDSR can be implemented as a fixed-value register.

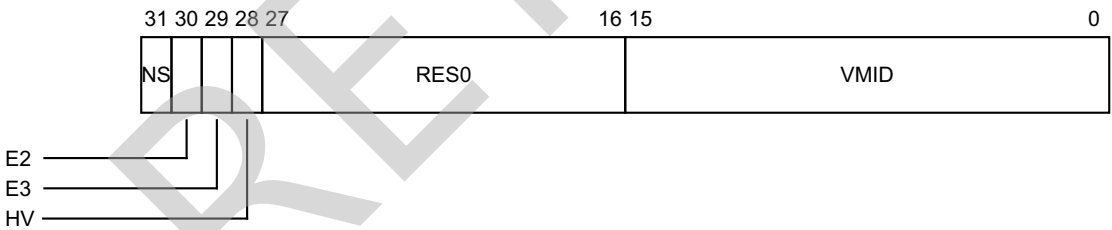
Attributes

In ARMv8.1, the format of this register differs depending on the value of [EDSCR.SC2](#).

Field descriptions

The EDVIDSR bit assignments are:

When *EDSCR.SC2* == 0:



This format applies in all ARMv8.0 implementations.

NS, bit [31]

Non-secure state sample. Indicates the Security state associated with the most recent [EDPCSR](#) sample.

If EL3 is not implemented, this bit has a fixed read-only value.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

E2, bit [30]

Exception level 2 status sample. Indicates whether the most recent [EDPCSR](#) sample was associated with EL2. If *EDVIDSR.NS* == 0, this bit is 0.

If EL2 is not implemented, this bit is RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

E3, bit [29]

Exception level 3 status sample. Indicates whether the most recent [EDPCSR](#) sample was associated with AArch64 EL3. If [EDVIDSR.NS](#) == 1 or the PE was in AArch32 state when [EDPCSR](#) was read, this bit is 0.

If EL3 is not implemented, this bit is RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

HV, bit [28]

[EDPCSR](#) high half valid. Indicates whether bits [63:32] of the most recent [EDPCSR](#) sample are valid. If [EDVIDSR.HV](#) == 0, the value of [EDPCSR](#)[63:32] is RAZ.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [27:16]

Reserved, RES0.

VMID, bits [15:0]

VMID sample. The VMID associated with the most recent [EDPCSR](#) sample. If [EDVIDSR.NS](#) == 0 or [EDVIDSR.E2](#) == 1, this field is RAZ.

If EL2 is not implemented, this field is RES0.

If EL2 is implemented and is using AArch64, the VMID is held in [VTTBR_EL2.VMID](#).

If EL2 is implemented and is using AArch32, the VMID is held in [VTTBR.VMID](#).

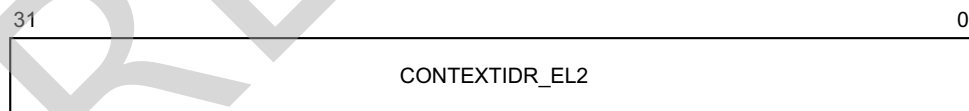
In ARMv8.0 the VMID is 8 bits.

In ARMv8.1 it is IMPLEMENTATION DEFINED whether the VMID is 8 bits or 16 bits.

If the implementation has an 8 bit VMID, then the upper 8 bits of this field are RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

When [EDSCR.SC2](#) == 1:



CONTEXTIDR_EL2, bits [31:0]

If the PC Sample in [EDPCSR](#) is from EL2 using AArch64, this register contains the sampled value of [CONTEXTIDR_EL2](#), captured on reading the low half of [EDPCSR](#).

Otherwise the value of this register is UNKNOWN.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the EDVIDSR:

EDVIDSR can be accessed through the external debug interface:

Component	Offset
Debug	0x0A8

RETIRED

D2.3 Performance Monitors registers

This section provides full descriptions of all of the Performance Monitors registers that are accessible through the external debug interface and are affected by the introduction of ARMv8.1, or by the changes to the Performance Monitors Extension introduced with ARMv8. See [General information about External debug register descriptions on page D2-706](#) for more information about the descriptions of these registers.

RETIRED

D2.3.1 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which common architectural and common microarchitectural feature events in the range 0x4000 to 0x401F are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	EPMAD	SLK	Default
Error	Error	Error	Error	RO	RO

Configurations

External register PMCEID2 is architecturally mapped to AArch64 System register [PMCEID0_ELO](#)[63:32].

External register PMCEID2[63:32] is architecturally mapped to AArch32 System register [PMCEID2](#).

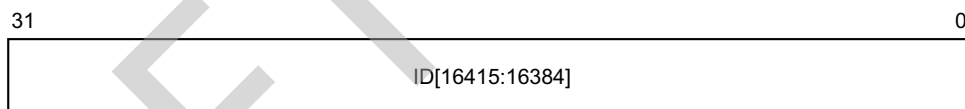
PMCEID2 is in the Core power domain.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

The PMCEID2 bit assignments are:



ID[16415:16384], bits [31:0]

PMCEID2[31:0] maps to common events 0x4000 to 0x401F. For a list of event numbers and descriptions, see .

For each bit:

0 The common event is not implemented.

1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Accessing the PMCEID2:

PMCEID2 can be accessed through the external debug interface:

Component	Offset
PMU	0xE28

D2.3.2 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which common architectural and common microarchitectural feature events in the range 0x4020 to 0x403F are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	EPMAD	SLK	Default
Error	Error	Error	Error	RO	RO

Configurations

External register PMCEID3 is architecturally mapped to AArch64 System register [PMCEID1_ELO](#)[63:32].

External register PMCEID3[63:32] is architecturally mapped to AArch32 System register [PMCEID3](#).

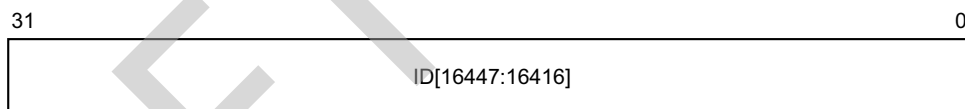
PMCEID3 is in the Core power domain.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

The PMCEID3 bit assignments are:



ID[16447:16416], bits [31:0]

PMCEID3[31:0] maps to common events 0x4020 to 0x403F. For a list of event numbers and descriptions, see .

For each bit:

0 The common event is not implemented.

1 The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

Accessing the PMCEID3:

PMCEID3 can be accessed through the external debug interface:

Component	Offset
PMU	0xE2C

D2.3.3 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	EPMA D	SLK	Default
Error	Error	Error	Error	RO	RW

Configurations

External register PMCR_EL0[6:0] is architecturally mapped to AArch32 System register [PMCR](#)[6:0].

External register PMCR_EL0[6:0] is architecturally mapped to AArch64 System register **PMCR_EL0**[6:0].

PMCR_EL0 is in the Core power domain. Some or all RW fields of this register have defined reset values. These apply on a Warm or Cold reset. The register is not affected by an External debug reset.

This register is only partially mapped to the internal **PMCR** System register. An external agent must use other means to discover the information held in **PMCR**[31:11], such as accessing **PMCFGR** and the ID registers.

Attributes

PMCR_EL0 is a 32-bit register.

Field descriptions

The PMCR EL0 bit assignments are:

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														</
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

Bits [31:11]

Reserved, RAZ/WI. Hardware must implement this as RAZ/WI. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

Bits [10:7]

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines which PMCCNTR_EL0 bit generates an overflow recorded by PMOVSr[31].

0	Cycle counter overflow on increment that changes PMCCNTR_EL0[31] from 1 to 0.
---	---

1 Cycle counter overflow on increment that changes PMCCNTR_EL0[63] from 1 to 0.

ARM deprecates use of PMCR EL0.LC = 0.

In an AArch64-only implementation, this field is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

DP, bit [5]

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

- 0 PMCCNTR_EL0, if enabled, counts when event counting is prohibited.
- 1 PMCCNTR_EL0 does not count when event counting is prohibited.

Event counting is prohibited when `ProfilingProhibited(IsSecure(),PSTATE.EL) == TRUE`.

When EL3 is not implemented, this field is RES0:

- In ARMv8.0.
- In ARMv8.1, only if EL2 is not implemented.

Otherwise this field is RW.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:

- A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
- 0 if the reset is into an Exception level that is using AArch32.

X, bit [4]

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

- 0 Do not export events.
- 1 Export events where not prohibited.

This field enables the exporting of events over an event bus to another device, for example to an OPTIONAL trace macrocell. If the implementation does not include such an event bus then this field is RAZ/WI, otherwise it is an RW field.

In an implementation that includes an event bus, no events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:

- A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
- 0 if the reset is into an Exception level that is using AArch32.

D, bit [3]

Clock divider. The possible values of this bit are:

- 0 When enabled, PMCCNTR_EL0 counts every clock cycle.
- 1 When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

In an AArch64-only implementation this field is RES0, otherwise it is an RW field. If `PMCR_EL0.LC == 1`, this bit is ignored and the cycle counter counts every clock cycle.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:

- A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
- 0 if the reset is into an Exception level that is using AArch32.

C, bit [2]

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

- 0 No action.

1 Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Resetting PMCCNTR_EL0 does not clear the PMCCNTR_EL0 overflow bit to 0.

P, bit [1]

Event counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset all event counters, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

Resetting the event counters does not clear any overflow bits to 0.

E, bit [0]

Enable. The possible values of this bit are:

0 All counters, including PMCCNTR_EL0, are disabled.

1 All counters are enabled by PMCNTESET_EL0.

This bit is RW.

When this register has an architecturally-defined reset value, this field resets to 0.

Accessing the PMCR_EL0:

PMCR_EL0 can be accessed through the external debug interface:

Component	Offset
PMU	0xE04

D2.3.4 **PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30**

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Usage constraints

This register is accessible as follows:

Off	DLK	OSLK	EPMA	SLK	Default
Error	Error	Error	Error	RO	RW

Configurations

External register PMEVTYPER<n>_EL0 is architecturally mapped to AArch64 System register [PMEVTYPER<n>_EL0](#).

External register PMEVTYPER<n>_EL0 is architecturally mapped to AArch32 System register [PMEVTYPER<n>](#).

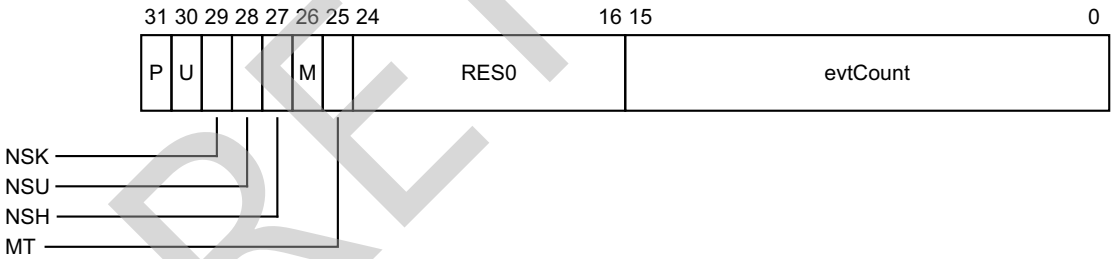
PMEVTYPER<n>_EL0 is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply on a Warm or Cold reset. The register is not affected by an External debug reset.

Attributes

PMEVTYPER<n>_EL0 is a 32-bit register.

Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:



P, bit [31]

Privileged filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

- 0 Count events in EL1.
- 1 Do not count events in EL1.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

U, bit [30]

User filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

- 0 Count events in EL0.
- 1 Do not count events in EL0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

NSK, bit [29]

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

NSU, bit [28]

Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

NSH, bit [27]

Non-secure EL2 (Hypervisor) filtering. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

0 Do not count events in EL2.

1 Count events in EL2.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

M, bit [26]

Secure EL3 filtering bit. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Secure EL3 are counted.

Otherwise, cycles in Secure EL3 are not counted.

Most applications can ignore this field and set its value to 0.

————— Note —————

This field is not visible in the AArch32 PMEVTYPER System register.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

MT, bit [25]

Multithreading. When the implementation is multi-threaded, the valid values for this bit are:

0 Count events only on controlling PE.

1 Count events from any PE with the same affinity at level 1 and above as this PE.

When the implementation is not multi-threaded, this bit is RES0.

————— Note —————

- An implementation is described as multi-threaded when the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. That is, the performance of PEs at the lowest affinity level is highly interdependent. On such an implementation, the value of MPIDR_EL1.MT, when read at the highest implemented Exception level, is 1.

- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [24:16]

Reserved, RES0.

evtCount, bits [15:0] (In ARMv8.1)

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event that is supported by the PE being programmed.

There are three ranges of event numbers:

- Event numbers in the range 0x000 to 0x03F are common architectural and microarchitectural events.
- Event numbers in the range 0x040 to 0x0BF are ARM recommended common architectural and microarchitectural events.
- Event numbers in the range 0x0C0 to 0x3FF are IMPLEMENTATION DEFINED events.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the event type:

- For the range 0x000 to 0x03F, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN. UNPREDICTABLE in this case means the event must not expose privileged information.

Note

UNPREDICTABLE means the event must not expose privileged information.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back from evtCount is UNKNOWN.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [15:10] (In ARMv8.0)

Reserved, RES0.

evtCount, bits [9:0] (In ARMv8.0)

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event that is supported by the PE being programmed.

There are three ranges of event numbers:

- Event numbers in the range 0x000 to 0x03F are common architectural and microarchitectural events.
- Event numbers in the range 0x040 to 0x0BF are ARM recommended common architectural and microarchitectural events.

- Event numbers in the range 0x0C0 to 0x3FF are IMPLEMENTATION DEFINED events.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the event type:

- For the range 0x000 to 0x03F, no events are counted, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED events, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN. UNPREDICTABLE in this case means the event must not expose privileged information.

Note

UNPREDICTABLE means the event must not expose privileged information.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back from evtCount is UNKNOWN.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the PMEVTYPER<n>_EL0:

PMEVTYPER<n>_EL0 can be accessed through the external debug interface:

Component	Offset
PMU	0x400 + 4n

Part E

Architectural Pseudocode

RETIRED

Chapter E1

ARMv8.1 Pseudocode

This chapter defines pseudocode that describes various features of the ARMv8.1 architecture, for operation in AArch64 and in AArch32 state, including a summary of the changes made by the introduction of ARMv8.1. It contains the following sections:

- *About the ARMv8.1 pseudocode chapter on page E1-746.*
- *Library pseudocode for AArch64 on page E1-747.*
- *Library pseudocode for AArch32 on page E1-807.*
- *Common library pseudocode on page E1-875.*

E1.1 About the ARMv8.1 pseudocode chapter

This chapter provides the complete architectural pseudocode chapter from the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, updated to take account of the changes introduced by ARMv8.1. [ARMv8.1 pseudocode changes](#) identifies the functions that are changed for ARMv8.1.

Note

The extensive cross-referencing in the ARM architectural pseudocode means it is more appropriate to include the full pseudocode library.

E1.1.1 ARMv8.1 pseudocode changes

The following functions have been updated in ARMv8.1:

AArch64

- [aarch64/debug/breakpoint/AArch64.BreakpointValueMatch](#) on page E1-747.
- [aarch64/debug/pmu/AArch64.CountEvents](#) on page E1-751.
- [aarch64/exceptions/takeexception/AArch64.TakeException](#) on page E1-762.
- [aarch64/translation/checks/AArch64.CheckPermission](#) on page E1-792.
- [aarch64/translation/walk/AArch64.TranslationTableWalk](#) on page E1-799.

AArch32

- [aarch32/debug/breakpoint/AArch32.BreakpointValueMatch](#) on page E1-808.
- [aarch32/debug/pmu/AArch32.CountEvents](#) on page E1-811.
- [aarch32/exceptions/takeexception/AArch32.EnterMode](#) on page E1-826.
- [aarch32/exceptions/takeexception/AArch32.EnterMonitorMode](#) on page E1-826.
- [aarch32/translation/checks/AArch32.CheckPermission](#) on page E1-860.

Shared

- [shared/debug/halting/DCPSInstruction](#) on page E1-881.
- [shared/debug/samplebasedprofiling/CreatePCSample](#) on page E1-888.
- [shared/debug/samplebasedprofiling/EDPCSRlo](#) on page E1-888.
- [shared/functions/system/IsInHost](#) on page E1-936.
- [shared/translation/translation/HasS2Translation](#) on page E1-947.
- [shared/translation/translation/S1TranslationRegime](#) on page E1-947.

E1.2 Library pseudocode for AArch64

E1.2.1 aarch64/debug

aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpt = 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, isbreakpt, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();

    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();

    match = value_match && state_match && enabled;

    return match;
```

aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// =====

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
```

```
// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking.)
if DBGBCR_EL1[n].E == '0' then return FALSE;

context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
type = DBGBCR_EL1[n].BT;

if ((type IN {'011x', '11xx'} && !HaveVirtHostExt()) || // Context matching
    type == '010x' || // Reserved
    (type != '0x0x' && !context_aware) || // Context matching
    (type == '1xxx' && !HaveEL(EL2))) then // EL2 extension
    (c, type) = ConstrainUnpredictableBits();
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (type == '0x0x');
match_vmid = (type == '10xx');
match_cid = (type == '001x');
match_cid1 = (type IN {'101x', 'x11x'});
match_cid2 = (type == '11xx');
linked = (type == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return FALSE;

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    if HaveAnyAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned.
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
    top = AddrTop(vaddress, PSTATE.EL);
    BVR_match = vaddress<top:2> == DBGBCR_EL1[n]<top:2> && byte_select_match;
elseif match_cid then
    if IsInHost() then
        BVR_match = (CONTEXTIDR_EL2 == DBGBCR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1 == DBGBCR_EL1[n]<31:0>);
elseif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1 == DBGBCR_EL1[n]<31:0>);

if match_vmid then
    if !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGBCR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGBCR_EL1[n]<47:32>;
    BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        !IsInHost() &&
        vmid == bvr_vmid);
elseif match_cid2 then
    BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
        DBGBCR_EL1[n]<63:32> == CONTEXTIDR_EL2);
```

```
bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

return match;
```

aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreaknt, boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreaknt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    if ((HMC:SSC:PxC) IN {'011xx', '100x0', '101x0', '11010', '11101', '1111x'}) || // Reserved
        (HMC == '0' && PxC == '00' && (!isbreaknt || !HaveAArch32EL(EL1))) || // Upr/Svc/Sys
        (SSC IN {'01', '10'} && !HaveEL(EL3)) || // No EL3
        (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL(EL3) && !HaveEL(EL2)) || // No EL3/EL2
        (HMC:SSC:PxC == '11100' && !HaveEL(EL2))) then // No EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && HMC == '1';
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    case PSTATE.EL of
        when EL3 priv_match = EL3_match;
        when EL2 priv_match = EL2_match;
        when EL1 priv_match = if ispriv || isbreaknt then EL1_match else EL0_match;
        when EL0 priv_match = EL0_match;

    case SSC of
        when '00' security_state_match = TRUE; // Both
        when '01' security_state_match = !IsSecure(); // Non-secure only
        when '10' security_state_match = IsSecure(); // Secure only
        when '11' security_state_match = TRUE; // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
        last_ctx_cmp = UInt(ID_AA64DFR0_EL1.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED return FALSE; // Disabled
                when Constraint_NONE linked = FALSE; // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(64) UNKNOWN;
```

```

        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

        return priv_match && security_state_match && (!linked || linked_match);

```

aarch64/debug/enables/AArch64.GenerateDebugExceptions

```

// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);

```

aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```

// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    if HaveEL(EL3) && secure then
        enabled = MDCR_EL3.SDD == '0' && from != EL3;
    else
        enabled = TRUE;

    target = if route_to_el2 then EL2 else EL1;
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';

    return enabled;

```

aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```

// AArch64.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = (PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1');
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMIIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;

```

aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====
// Return TRUE if counter "n" should count its event.

boolean AArch64.CountEvents(integer n)
    assert(n == 31 || n < UInt(PMCR_EL0.N));

    // Event counting is disabled in Debug state
    debug = Halted();

    if HaveEL(EL2) then
        E = (if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME);
    else
        E = PMCR_EL0.E;
    enabled = (E == '1' && PMCNTENSET_EL0<n> == '1');

    // Event counting might be prohibited
    prohibited = AArch64.ProfilingProhibited(IsSecure(), PSTATE.EL);
    if PSTATE.EL == EL2 && HaveHPMDExt() && (n < UInt(MDCR_EL2.HPMN) || n == 31) then
        prohibited = (MDCR_EL2.HPMD == '1' && !ExternalSecureNoninvasiveDebugEnabled());
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M} bits
    filter = (if n == 31 then PMCCFILTR_EL0<31:26> else PMEVTYPER_EL0[n]<31:26>);

    M = if !HaveEL(EL3) then '0' else (filter<5> EOR filter<0>);
    H = if !HaveEL(EL2) then '0' else filter<1>;
    P = filter<5>; U = filter<4>;
    if !IsSecure() && HaveEL(EL3) then
        P = P EOR filter<3>; U = U EOR filter<2>;

    case PSTATE.EL of
        when EL0 filtered = U == '1';
        when EL1 filtered = P == '1';
        when EL2 filtered = H == '0';
        when EL3 filtered = M == '1';

    return !debug && enabled && !prohibited && !filtered;
```

aarch64/debug/pmu/AArch64.ProfilingProhibited

```
// AArch64.ProfilingProhibited()
// =====
// Determine whether event counting is prohibited in the current state.

boolean AArch64.ProfilingProhibited(boolean secure, bits(2) el)

    // Events are always counted in Non-secure state.
    if !secure then return FALSE;

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    if !HaveEL(EL3) then return FALSE;

    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    if MDCR_EL3.SPME == '1' then return FALSE;

    // * Allowed by the IMPLEMENTATION DEFINED authentication interface
    if ExternalSecureNoninvasiveDebugEnabled() then return FALSE;

    return TRUE;
```

aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el; PSTATE.nRW = '0'; PSTATE.SP = '1';

    SPSR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000'; PSTATE.T = '0'; // PSTATE.J is RES0
    if HavePANExt() && (PSTATE.EL == EL1 || IsInHost()) && SCTLRL.SPAN == '0' then
        PSTATE.PAN = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.
    EndOfInstruction();
```

aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)

    top = AddrTop(vaddress, PSTATE.EL);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    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();
            bottom = 3; // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE; // Disabled
            when Constraint_NONE mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
```



```
// If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
    WVR_match = ConstrainUnpredictableBool();
else
    WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

return WVR_match && byte_select_match;
```

aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

E1.2.2 aarch64/exceptions

aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elseif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception type, FaultRecord fault, bits(64) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type IN {Exception_DataAbort, Exception_Watchpoint};

    exception.syndrome = FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPValid(fault) then
        exception.ipvalid = TRUE;
        exception.ipaddress = fault.ipaddress;
    else
        exception.ipvalid = FALSE;

    return exception;
```

aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// =====

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure()) && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' || IsSecondStage(fault));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elseif 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);
```

aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// =====

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure()) && PSTATE.EL IN {EL0, EL1} &&
```

```
(HCR_EL2.TGE == '1' || IsSecondStage(fault)));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

if PSTATE.EL == EL3 || route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_PCAlignment);
exception.vaddress = ThisInstrAddr();

if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SPAlignment);

if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalFIQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr();
```

```

vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);

if route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elseif 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);

```

aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```

// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);

if route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elseif 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);

```

aarch64/exceptions/asynch/AArch64.TakePhysicalSystemErrorException

```

// AArch64.TakePhysicalSystemErrorException()
// =====

AArch64.TakePhysicalSystemErrorException(boolean syndrome_valid, bits(24) syndrome)

route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;

exception = ExceptionSyndrome(Exception_SError);
if syndrome_valid then
    exception.syndrome<24> = '1';
    exception.syndrome<23:0> = syndrome;
else
    exception.syndrome<24> = '0';

if PSTATE.EL == EL3 || route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elseif 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);

```

aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualFIQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualIRQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualSystemErrorException

```
// AArch64.TakeVirtualSystemErrorException()
// =====

AArch64.TakeVirtualSystemErrorException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    if boolean IMPLEMENTATION_DEFINED "Virtual System Error syndrome valid" then
        exception.syndrome<24> = '1';
        exception.syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "Virtual System Error syndrome";

    HCR_EL2.VSE = '0';
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =====

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
```

```
vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

if PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// =====

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
```

```

assert HaveEL(EL2) && !IsSecure() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/debug/AArch64.WatchpointException

```

// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/exceptions/AArch64.ExceptionClass

```

// AArch64.ExceptionClass()
// =====
// Return the Exception Class and Instruction Length fields for reported in ESR

(integer, bit) AArch64.ExceptionClass(Exception type, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    case type of
        when Exception_Uncategorized    ec = 0x00; il = '1';
        when Exception_WFxTrap          ec = 0x01;
        when Exception_CP15RRTTrap      ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap      ec = 0x04; assert from_32;
        when Exception_CP14RRTTrap      ec = 0x05; assert from_32;
        when Exception_CP14DRTTrap      ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap          ec = 0x08;
        when Exception_CP14RRTTrap      ec = 0x0C; assert from_32;
        when Exception_IllegalState      ec = 0x0E; il = '1';
        when Exception_SupervisorCall    ec = 0x11;
        when Exception_HypervisorCall    ec = 0x12;
        when Exception_MonitorCall       ec = 0x13;
        when Exception_SystemRegisterTrap ec = 0x18; assert !from_32;
        when Exception_InstructionAbort  ec = 0x20; il = '1';
        when Exception_PCAlignment       ec = 0x22; il = '1';
        when Exception_DataAbort         ec = 0x24;
        when Exception_SPAlignment       ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPtrappedException ec = 0x28;
        when Exception_SError            ec = 0x2F; il = '1';
        when Exception_Breakpoint        ec = 0x30; il = '1';
        when Exception_SoftwareStep      ec = 0x32; il = '1';

```

```

        when Exception_Watchpoint          ec = 0x34; i1 = '1';
        when Exception_SoftwareBreakpoint ec = 0x38;
        when Exception_VectorCatch         ec = 0x3A; i1 = '1'; assert from_32;
        otherwise                          Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,i1);

```

aarch64/exceptions/exceptions/AArch64.ReportException

```

// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception type = exception.type;

    (ec,i1) = AArch64.ExceptionClass(type, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        i1 = '1';

    ESR[target_el] = ec<5:0>:i1:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<39:4> = exception.ipaddress<47:12>;
        else
            HPFAR_EL2<39:4> = bits(36) UNKNOWN;

    return;

```

aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```

// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);

```

aarch64/exceptions/exceptions/AArch64.TakeReset

```

// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then

```



```

        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';           // Select stack pointer
    PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0';           // Clear software step bit
    PSTATE.IL = '0';           // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elseif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax(>>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType_UNKNOWN);

```

aarch64/exceptions/ieee754/AArch64.FPTrappedException

```

// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    exception.syndrome<23> = '1';           // TFV
    if is_ase then exception.syndrome<10:8> = element<2:0>; // VECITR
    exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    route_to_el2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/syscalls/AArch64.CallHypervisor

```

// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

```

```

if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();

bits(64) preferred_exception_return = NextInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_HypervisorCall);
exception.syndrome<15:0> = immediate;

if PSTATE.EL == EL3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```

// AArch64.CallSecureMonitor()
// =====

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/syscalls/AArch64.CallSupervisor

```

// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/takeexception/AArch64.TakeException

```

// AArch64.TakeException()
// =====
// Take an exception to an Exception Level using AArch64.

```

```

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                      bits(64) preferred_exception_return, integer vect_offset)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if !IsSecure() && HaveEL(EL2) then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elseif 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);

    elseif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    spsr = GetPSRFromPSTATE();

    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el; PSTATE.nRW = '0'; PSTATE.SP = '1';

    SPSR[] = spsr;
    ELR[] = preferred_exception_return;

    PSTATE.SS = '0';
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then
        PSTATE.IT = '00000000'; PSTATE.T = '0'; // Coming from AArch32
        // PSTATE.J is RES0
    if HavePANExt() && (PSTATE.EL == EL1 || IsInHost()) && SCTLR[].SPAN == '0' then
        PSTATE.PAN = '1';
    BranchTo(VBAR[] + vect_offset, BranchType_EXCEPTION);
    EndOfInstruction();

```

aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, bits(32) aarch32_instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(aarch32_instr);

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception_FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception_CP14RTTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RTTTrap);
            otherwise  Unreachable();
        iss<19:17> = instr<7:5>;    // opc2
        iss<16:14> = instr<23:21>;  // opc1
        iss<13:10> = instr<19:16>;  // CRn
        iss<9:5>   = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>; // Rt
        iss<4:1>   = instr<3:0>;    // CRm
    elseif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception_CP14RRTTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RRTTTrap);
            otherwise  Unreachable();
        iss<19:16> = instr<7:4>;    // opc1
        iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rt2
        iss<9:5>   = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>; // Rt
        iss<4:1>   = instr<3:0>;    // CRm
    elseif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        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 // Literal addressing
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
        else
            iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
            iss<3>   = '0';
        else
            Unreachable();
        iss<0> = instr<20>;          // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0> = iss;

    return exception;
```

aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
```

```

route_to_el2 = (target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1');

if route_to_el2 then
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    exception = ExceptionSyndrome(Exception_AdvSIMDFAccessTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

return;

```

aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccess

```

// AArch64.CheckAArch32SystemAccess()
// =====
// Check AArch32 System register access instruction for enables and disables

AArch64.CheckAArch32SystemAccess(bits(32) instr)
    cp_num = UInt(instr<11:8>);
    assert cp_num IN {14,15};

    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then // MRC/MCR
        cprt = TRUE; cpdt = FALSE; nreg = 1;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn = UInt(instr<19:16>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:21> == '1100010' then // MRRC/MCRR
        cprt = TRUE; cpdt = FALSE; nreg = 2;
        opc1 = UInt(instr<7:4>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then // LDC/STC
        cprt = FALSE; cpdt = TRUE; nreg = 0;
        opc1 = 0;
        CRn = UInt(instr<15:12>);
    else
        allocated = FALSE;

    //
    // Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
    // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
    if cp_num == 14 then
        // LDC and STC only supported for c5 in CP14 encoding space
        if cpdt && CRn != 5 then
            allocated = FALSE;
        else
            // Coarse-grained decode of CP14 based on opc1 field
            case opc1 of
                when 0    allocated = CP14DebugInstrDecode(instr);
                when 1    allocated = CP14TraceInstrDecode(instr);
                when 7    allocated = CP14JazelleInstrDecode(instr); // JIDR only
                otherwise allocated = FALSE; // All other values are unallocated

    elseif cp_num == 15 then
        // LDC and STC not supported in CP15 encoding space
        if !cprt then
            allocated = FALSE;
        else
            allocated = CP15InstrDecode(instr);

    // Coarse-grain traps to EL2 have a higher priority than Undefined Instruction
    if AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
        // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
        // Non-secure User mode is UNDEFINED when the trap is disabled, then it is

```

```

// IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
// when the trap is enabled.
if PSTATE.EL == EL0 && !IsSecure() && !allocated then
    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
        UNDEFINED;
    AArch64.AArch32SystemAccessTrap(EL2, instr);

else
    allocated = FALSE;

if !allocated then
    UNDEFINED;

// If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
AArch64.CheckAArch32SystemAccessTraps(instr);

return;

```

aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessTraps

// Check for configurable disables or traps to a higher EL of an AArch32 System register access.
AArch64.CheckAArch32SystemAccessTraps(bits(32) instr);

aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```

// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

// Check for coarse-grained Hyp traps
if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
    // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
    major = if nreg == 1 then CRn else CRm;
    if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
        return TRUE;

    // Check for MRC and MCR disabled by HCR_EL2.TIDCP
    if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
        ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
         (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
         (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
        return TRUE;

return FALSE;

```

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```

// AArch64.CheckFPAdvSIMDEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPAdvSIMDEnabled()
if PSTATE.EL IN {EL0, EL1} then
    // Check if access disabled in CPACR_EL1
    case CPACR[].FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

AArch64.CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3

```

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()

    if HaveEL(EL2) && !IsSecure() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = !(PSTATE.EL == EL1 && HCR_EL2.TGE == '1');
                when '01' disabled = (PSTATE.EL == EL0 && HCR_EL2.TGE == '1');
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

aarch64/exceptions/traps/AArch64.CheckForSMCTrap

```
// AArch64.CheckForSMCTrap()
// =====
// Check for trap on SMC instruction

AArch64.CheckForSMCTrap(bits(16) imm)

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.CheckForWFXTrap

```
// AArch64.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFXTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1 trap = (if is_wfe then SCTLRL.nTWE else SCTLRL.nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

    if trap then
        AArch64.WFXTrap(target_el, is_wfe);
```

aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
```

```

if PSTATE.IL == '1' then
    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_IllegalState);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/traps/AArch64.MonitorModeTrap

```

// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/traps/AArch64.SystemRegisterTrap

```

// AArch64.SystemRegisterTrap()
// =====
// Trapped system register access other than due to CPTR_EL2 and CPACR_EL1

AArch64.SystemRegisterTrap(bits(2) target_el, bits(2) op0, bits(3) op2, bits(3) op1, bits(4) crn,
                           bits(5) rt, bits(4) crm, bit dir)
    assert UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
    exception.syndrome<21:20> = op0;
    exception.syndrome<19:17> = op2;
    exception.syndrome<16:14> = op1;
    exception.syndrome<13:10> = crn;
    exception.syndrome<9:5> = rt;
    exception.syndrome<4:1> = crm;
    exception.syndrome<0> = dir;

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

```

aarch64/exceptions/traps/AArch64.UndefinedFault

```

// AArch64.UndefinedFault()
// =====

AArch64.UndefinedFault()

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

```



```
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_Uncategorized);

if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elseif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

E1.2.3 aarch64/functions

aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// =====

FaultRecord AArch64.CreateFaultRecord(Fault type, bits(48) ipaddress,
                                       integer level, AccType acctype, boolean write, bit extflag,
                                       boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.type = type;
    fault.domain = bits(4) UNKNOWN; // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN; // Not used from AArch64
    fault.ipaddress = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
```

```

    fault.s2fs1walk = s2fs1walk;

    return fault;

```

aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```

// AArch64.ExclusiveMonitorsPass()
// =====

// Return TRUE if the Exclusive Monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;

```

aarch64/functions/exclusive/AArch64.IsExclusiveVA

```

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

```

aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);

```

aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// =====

// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// =====

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1, sign1, value1) = FPUnpack(op1, FPCR);
    (type2, sign2, value2) = FPUnpack(op2, FPCR);
    (done, result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {32, 64};
```

```

bits(N) result;
op1 = FPNeg(op1);
(type1,sign1,value1) = FPUnpack(op1, FPCR);
(type2,sign2,value2) = FPUnpack(op2, FPCR);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo('0');
    elseif inf1 || inf2 then
        result = FPinfinity(sign1 EOR sign2);
    else
        // Fully fused multiply-add
        result_value = 2.0 + (value1 * value2);
        if result_value == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign);
        else
            result = FPRound(result_value, FPCR);
return result;

```

aarch64/functions/memory/AArch64.CheckAlignment

```

// AArch64.CheckAlignment()
// =====

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                                boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED };
    vector = acctype == AccType_VEC;
    check = (atomic || ordered || SCTLRL.A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;

```

aarch64/functions/memory/AArch64.MemSingle

```

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle(bits(64) address, integer size, AccType acctype, boolean wasaligned)
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

```

```
// Memory array access
value = _Mem[memaddrdesc, size, acctype];
return value;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    _Mem[memaddrdesc, size, acctype] = value;
    return;
```

aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];

    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

aarch64/functions/memory/Mem

```
// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    atomic = (aligned && !(acctype IN {AccType_VEC, AccType_VECSTREAM})) || size == 1;
```

```

if !atomic then
    assert size > 1;
    value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    if !aligned then
        c = ConstrainUnpredictable();
        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];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
integer i;
boolean iswrite = TRUE;

if BigEndian() then
    value = BigEndianReverse(value);

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
atomic = (aligned && !(acctype IN {AccType_VEC, AccType_VECSTREAM})) || size == 1;

if !atomic then
    assert size > 1;
    AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    if !aligned then
        c = ConstrainUnpredictable();
        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>;
    else
        AArch64.MemSingle[address, size, acctype, aligned] = value;
    return;

```

aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```

// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32(); // Always called from AArch32 state before entering AArch64 state

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elseif PSTATE.EL IN {EL0, EL1} && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        first = 0; last = 30; include_R15 = FALSE;

```

```

else
    first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool() then
            _R[n]<63:32> = Zeros();

    return;

```

aarch64/functions/registers/AArch64.ResetGeneralRegisters

```

// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;

```

aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```

// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;

```

aarch64/functions/registers/AArch64.ResetSpecialRegisters

```

// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;

```

aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

aarch64/functions/registers/PC

```
// PC - non-assignment form
// =====
// Read program counter.

bits(64) PC[]
return _PC;
```

aarch64/functions/registers/SP

```
// SP[] - assignment form
// =====
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
assert width IN {32,64};
if PSTATE.SP == '0' then
    SP_EL0 = ZeroExtend(value);
else
    case PSTATE.EL of
        when EL0 SP_EL0 = ZeroExtend(value);
        when EL1 SP_EL1 = ZeroExtend(value);
        when EL2 SP_EL2 = ZeroExtend(value);
        when EL3 SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// =====
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
assert width IN {8,16,32,64};
if PSTATE.SP == '0' then
    return SP_EL0<width-1:0>;
else
    case PSTATE.EL of
        when EL0 return SP_EL0<width-1:0>;
        when EL1 return SP_EL1<width-1:0>;
        when EL2 return SP_EL2<width-1:0>;
        when EL3 return SP_EL3<width-1:0>;
```

aarch64/functions/registers/V

```
// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
assert n >= 0 && n <= 31;
assert width IN {8,16,32,64,128};
_V[n] = ZeroExtend(value);
return;

// V[] - non-assignment form
// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
```



```
assert n >= 0 && n <= 31;
assert width IN {8,16,32,64,128};
return _V[n]<width-1:0>;
```

aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =====
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of the register;
// part 1 returns only the top 64 bits of the register.

bits(width) Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width == 64;
        return _V[n]<127:64>;

// Vpart[] - assignment form
// =====
// Write 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 64 bits of the register.

Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<127:64> = value<63:0>;
```

aarch64/functions/registers/X

```
// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// =====

CNTKCTLType CNTKCTL[]
    if IsInHost() then
        return CNTKCTL_EL2;
    return CNTKCTL_EL1;
```

aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =====

CPACRType CPACR[]
    if IsInHost() then
        return CPTR_EL2;
    return CPACR_EL1;
```

aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// =====

bits(64) ELR[bits(2) e1]
    bits(64) r;
    case e1 of
        when EL1 r = ELR_EL1;
        when EL2 r = ELR_EL2;
        when EL3 r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// =====

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =====

ELR[bits(2) e1] = bits(64) value
    bits(64) r = value;
    case e1 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;
```

aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// =====

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1 r = ESR_EL1;
        when EL2 r = ESR_EL2;
        when EL3 r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// =====

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =====

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1 ESR_EL1 = r;
        when EL2 ESR_EL2 = r;
        when EL3 ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =====

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// =====

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = FAR_EL1;
        when EL2 r = FAR_EL2;
        when EL3 r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// =====

bits(64) FAR[]
    return FAR[S1TranslationRegime()];
```

```
// FAR[] - assignment form
// =====

FAR[bits(2) regime] = bits(64) value
bits(64) r = value;
case regime of
    when EL1 FAR_EL1 = r;
    when EL2 FAR_EL2 = r;
    when EL3 FAR_EL3 = r;
    otherwise Unreachable();
return;

// FAR[] - assignment form
// =====

FAR[] = bits(64) value
FAR[S1TranslationRegime()] = value;
return;
```

aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// =====

MAIRType MAIR[bits(2) regime]
bits(64) r;
case regime of
    when EL1 r = MAIR_EL1;
    when EL2 r = MAIR_EL2;
    when EL3 r = MAIR_EL3;
    otherwise Unreachable();
return r;

// MAIR[] - non-assignment form
// =====

MAIRType MAIR[]
return MAIR[S1TranslationRegime()];
```

aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[bits(2) regime]
bits(32) r;
case regime of
    when EL1 r = SCTLR_EL1;
    when EL2 r = SCTLR_EL2;
    when EL3 r = SCTLR_EL3;
    otherwise Unreachable();
return r;

// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[]
return SCTLR[S1TranslationRegime()];
```

aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// =====

bits(64) VBAR[bits(2) regime]
bits(64) r;
case regime of
    when EL1 r = VBAR_EL1;
    when EL2 r = VBAR_EL2;
    when EL3 r = VBAR_EL3;
    otherwise Unreachable();
return r;

// VBAR[] - non-assignment form
// =====

bits(64) VBAR[]
return VBAR[S1TranslationRegime()];
```

aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a SIMD or floating-point
// register is trapped under the current configuration. Returns a boolean which
// is TRUE if trapping occurs, plus a binary value that specifies the Exception
// level trapped to.
(boolean, bits(2)) AArch64.CheckAdvSIMDFPSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn,
bits(4) crm, bits(3) op2, bit read);
```

aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// =====

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt,
bit read)
// Checks if an AArch64 MSR, MRS or SYS instruction is UNALLOCATED or trapped at the current
// exception level, security state and configuration, based on the opcode's encoding.
boolean unallocated = FALSE;
boolean need_secure = FALSE;
bits(2) min_EL;

// Check for traps by HCR_EL2.TIDCP
if HaveEL(EL2) && !IsSecure() && HCR_EL2.TIDCP == 1 && op0 == 'x1' && crn == '1x11' then
    // At Non-secure EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
    // register access instructions with reserved encodings are trapped to EL2 or UNDEFINED
    rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
    if PSTATE.EL == EL0 && rcs_el0_trap then
        AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);
    elseif PSTATE.EL == EL1 then
        AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);

// Check for unallocated encodings
case op1 of
    when '00x', '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !HaveVirtHostExt() then UnallocatedEncoding();
```

```

        min_EL = EL2;
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
        need_secure = TRUE;
    if UInt(PSTATE.EL) < UInt(min_EL) then
        UnallocatedEncoding();
    elsif need_secure && !IsSecure() then
        UnallocatedEncoding();
    elsif AArch64.CheckUnallocatedSystemAccess(op0, op1, crn, crm, op2, read) then
        UnallocatedEncoding();

    // Check for traps on accesses to SIMD or floating-point registers
    (take_trap, target_el) = AArch64.CheckAdvSIMDFPSystemRegisterTraps(op0, op1, crn, crm, op2);
    if take_trap then
        AArch64.AdvSIMDFPAccessTrap(target_el);

    // Check for traps on access to all other system registers
    (take_trap, target_el) = AArch64.CheckSystemRegisterTraps(op0, op1, crn, crm, op2, read);
    if take_trap then
        AArch64.SystemRegisterTrap(target_el, op0, op2, op1, crn, rt, crm, read);

```

aarch64/functions/system/AArch64.CheckSystemRegisterTraps

```

// Checks if an AArch64 MSR, MRS or SYS instruction on a system register is trapped
// under the current configuration. Returns a boolean which is TRUE if trapping
// occurs, plus a binary value that specifies the Exception level trapped to.
(boolean, bits(2)) AArch64.CheckSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
bits(3) op2, bit read);

```

aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess

```

// Checks if an AArch64 MSR, MRS or SYS instruction is unallocated under the current
// configuration.
boolean AArch64.CheckUnallocatedSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3)
op2, bit read);

```

aarch64/functions/system/AArch64.SysInstr

```

// Execute a system operation with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);

```

aarch64/functions/system/AArch64.SysInstrWithResult

```

// Execute a system operation with read (result operand).
// Returns the result of the operation.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);

```

aarch64/functions/system/AArch64.SysRegRead

```

// Read from a system register and return the contents of the register.
bits(64) System_Get(integer op0, integer op1, integer crn, integer crm, integer op2);

```

aarch64/functions/system/AArch64.SysRegWrite

```

// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);

```

E1.2.4 aarch64/instrs

aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

// Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
SetPSTATEFromSPSR(spsr);
ClearExclusiveLocal(ProcessorID());
EventRegisterSet();

if spsr<4> == '1' then
    // For an attempted to change to AArch32 state, align PC[1:0] according
    // to the target instruction set state. If the exception return is illegal,
    // it is IMPLEMENTATION DEFINED whether this alignment takes place.
    align_pc = boolean IMPLEMENTATION_DEFINED "Align PC on illegal exception return";
    if PSTATE.IL == '0' || align_pc then
        if spsr<5> == '1' then // T32
            new_pc = Align(new_pc, 2);
        else // A32
            new_pc = Align(new_pc, 4);

    // If the return was illegal, the 32 MSBs of the target PC might be zeroed
    if PSTATE.IL == '1' && ConstrainUnpredictableBool() then
        new_pc<63:32> = Zeros();

if UsingAArch32() then
    // 32 most significant bits are ignored
    BranchTo(new_pc<31:0>, BranchType_UNKNOWN);
else
    // For an illegal exception return it is IMPLEMENTATION DEFINED whether the return is
    // to the Exception level indicated by the SPSR, or to the Exception level
    // in which the exception return was executed.
    el_from_spsr = boolean IMPLEMENTATION_DEFINED "EL from SPSR on illegal exception return";
    target_el = PSTATE.EL;
    if PSTATE.IL == '1' && el_from_spsr then
        (-, target_el) = ELFromSPSR(spsr);
    new_pc = BranchAddr(new_pc, target_el);
    BranchToAddr(new_pc, BranchType_ERET);
```

aarch64/instrs/countop/CountOp

```
enumeration CountOp {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
case op of
    when '000' return ExtendType_UXTB;
    when '001' return ExtendType_UXTH;
    when '010' return ExtendType_UXTW;
    when '011' return ExtendType_UXTX;
    when '100' return ExtendType_SXTB;
    when '101' return ExtendType_SXTH;
    when '110' return ExtendType_SXTW;
    when '111' return ExtendType_SXTX;
```

aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType type, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case type of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SCTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE; len = 8;
        when ExtendType_UXTH unsigned = TRUE; len = 16;
        when ExtendType_UXTW unsigned = TRUE; len = 32;
        when ExtendType_UCTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SCTX,
    ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UCTX};
```

aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
    FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUUnaryOp

```
enumeration FPUUnaryOp {FPUUnaryOp_ABS, FPUUnaryOp_MOV,
    FPUUnaryOp_NEG, FPUUnaryOp_SQRT};
```

aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
    FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF};
```

aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
```



```
// must not match UBFIZ/SBFIX alias
if UInt(imms) < UInt(immr) then
    return FALSE;

// must not match LSR/ASR/LSL alias (imms == 31 or 63)
if imms == sf:'11111' then
    return FALSE;

// must not match UTX/SXT alias
if immr == '000000' then
    // must not match 32-bit UTX[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;
```

aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// =====

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then ReservedValue();
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        ReservedValue();

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    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);
```

aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != '1xxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =====
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;
```

aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType type, integer amount)
    bits(N) result = X[reg];
    case type of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP};
```

aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;           // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;          // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;         // PST: prepare for store
        when '11' return;                        // unallocated hint
    target = UInt(prfop<2:1>);                    // target cache level
    stream = (prfop<0> != '0');                  // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp {MemBarrierOp_DSB, MemBarrierOp_DMB, MemBarrierOp_ISB};
```

aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {SystemHintOp_NOP, SystemHintOp_YIELD,
    SystemHintOp_WFE, SystemHintOp_WFI,
    SystemHintOp_SEV, SystemHintOp_SEVL};
```

aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
    PSTATEField_PAN, // ARMv8.1
    PSTATEField_SP
};
```

aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =====

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
case op1:CRn:CRm:op2 of
    when '000 0111 1000 000' return Sys_AT; // S1E1R
    when '100 0111 1000 000' return Sys_AT; // S1E2R
    when '110 0111 1000 000' return Sys_AT; // S1E3R
    when '000 0111 1000 001' return Sys_AT; // S1E1W
    when '100 0111 1000 001' return Sys_AT; // S1E2W
    when '110 0111 1000 001' return Sys_AT; // S1E3W
    when '000 0111 1000 010' return Sys_AT; // S1E0R
    when '000 0111 1000 011' return Sys_AT; // S1E0W
    when '100 0111 1000 100' return Sys_AT; // S12E1R
    when '100 0111 1000 101' return Sys_AT; // S12E1W
    when '100 0111 1000 110' return Sys_AT; // S12E0R
    when '100 0111 1000 111' return Sys_AT; // S12E0W
    when '011 0111 0100 001' return Sys_DC; // ZVA
    when '000 0111 0110 001' return Sys_DC; // IVAC
    when '000 0111 0110 010' return Sys_DC; // ISW
    when '011 0111 0101 001' return Sys_DC; // CVAC
    when '000 0111 0101 010' return Sys_DC; // CSW
    when '011 0111 0101 001' return Sys_DC; // CVAU
    when '011 0111 1110 001' return Sys_DC; // CIVAC
    when '000 0111 1110 010' return Sys_DC; // CISW
    when '000 0111 0001 000' return Sys_IC; // IALLUIS
    when '000 0111 0101 000' return Sys_IC; // IALLU
    when '011 0111 0101 001' return Sys_IC; // IVAU
    when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
    when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
    when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
    when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
    when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
    when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
    when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
    when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
    when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
    when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
    when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
    when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
    when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
    when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
    when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
    when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
    when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
    when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
    when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
    when '100 1000 0111 000' return Sys_TLBI; // ALLE2
    when '110 1000 0111 000' return Sys_TLBI; // ALLE3
    when '000 1000 0111 001' return Sys_TLBI; // VAE1
    when '100 1000 0111 001' return Sys_TLBI; // VAE2
    when '110 1000 0111 001' return Sys_TLBI; // VAE3
    when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
    when '000 1000 0111 011' return Sys_TLBI; // VAAE1
    when '100 1000 0111 100' return Sys_TLBI; // ALLE1
    when '000 1000 0111 101' return Sys_TLBI; // VALE1
    when '100 1000 0111 101' return Sys_TLBI; // VALE2
    when '110 1000 0111 101' return Sys_TLBI; // VALE3
    when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
    when '000 1000 0111 111' return Sys_TLBI; // VAALE1
return Sys_SYS;
```

aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp {CompareOp_GT, CompareOp_GE, CompareOp_EQ,  
CompareOp_LE, CompareOp_LT};
```

aarch64/instrs/vector/crypto/enabled/CheckCryptoEnabled64

```
// CheckCryptoEnabled64()  
// =====  
  
CheckCryptoEnabled64()  
    AArch64.CheckFPAdvSIMDEnabled();  
    return;
```

aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,  
ImmediateOp_ORR, ImmediateOp_BIC};
```

aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()  
// =====  
  
bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)  
    integer half;  
    bits(esize) hi;  
    bits(esize) lo;  
    bits(esize) result;  
  
    if N == esize then  
        return input;  
  
    half = N DIV 2;  
    hi = Reduce(op, input<N-1:half>, esize);  
    lo = Reduce(op, input<half-1:0>, esize);  
  
    case op of  
        when ReduceOp_FMINNUM  
            result = FPMinNum(lo, hi, FPCR);  
        when ReduceOp_FMAXNUM  
            result = FPMaXNum(lo, hi, FPCR);  
        when ReduceOp_FMIN  
            result = FPMin(lo, hi, FPCR);  
        when ReduceOp_FMAX  
            result = FPMaX(lo, hi, FPCR);  
        when ReduceOp_FADD  
            result = FPAdd(lo, hi, FPCR);  
        when ReduceOp_ADD  
            result = lo + hi;  
  
    return result;
```

aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,  
ReduceOp_FMIN, ReduceOp_FMAX,  
ReduceOp_FADD, ReduceOp_ADD};
```

E1.2.5 aarch64/translation

aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// =====
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                             bits(48) ipaddress, integer level,
                                             AccType acctype, boolean iswrite, boolean secondstage,
                                             boolean s2fs1walk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, level, acctype, iswrite,
                                                  secondstage, s2fs1walk);
    else
        addrdesc.memattrs.type = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
```

aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// =====
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield<7:4> == '0000' then // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType_nGnRnE;
            when '0100' memattrs.device = DeviceType_nGnRE;
            when '1000' memattrs.device = DeviceType_nGRE;
            when '1100' memattrs.device = DeviceType_GRE;
            otherwise Unreachable(); // Reserved, handled above

    elsif attrfield<3:0> != '0000' then // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
```

```

else
    Unreachable();
    // Reserved, handled above

return MemAttrDefaults(memattrs);

```

aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```

// AArch64.TranslateAddressS1Off()
// =====
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

```

```

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

```

```

    TLBRecord result;

```

```

    Top = AddrTop(vaddress, PSTATE.EL);
    if !IsZero(vaddress < Top:PAMax()) then
        level = 0;
        ipaddress = bits(48) UNKNOWN;
        secondstage = FALSE;
        s2fslwalk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fslwalk);
    return result;

```

```

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

```

```

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB; // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elseif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
            result.addrdesc.memattrs.shareable = TRUE;
            result.addrdesc.memattrs.outershareable = TRUE;

```

```

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

```

```

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

```

```

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

```

```

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;

```

```

result.level = integer UNKNOWN;
result.blocksize = integer UNKNOWN;
result.addrdesc.paddress.physicaladdress = vaddress<47:0>;
result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
result.addrdesc.fault = AArch64.NoFault();

return result;

```

aarch64/translation/checks/AArch64.CheckPermission

```

// AArch64.CheckPermission()
// =====
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                     bit NS, AccType acctype, boolean iswrite)
assert !ELUsingAArch32(S1TranslationRegime());

wxn = SCTLRL[0].WXN == '1';

if PSTATE.EL IN {EL0, EL1} || IsInHost() then
    priv_r = TRUE;
    priv_w = perms.ap<2> == '0';
    user_r = perms.ap<1> == '1';
    user_w = perms.ap<2:1> == '01';

    if (HavePANExt() && PSTATE.PAN == '1' && user_r && PSTATE.EL != EL0 &&
        !(acctype IN {AccType_DC, AccType_AT, AccType_UNPRIV, AccType_IFETCH})) then
        priv_r = FALSE; priv_w = FALSE;

    user_xn = perms.xn == '1' || (user_w && wxn);
    priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;
    ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

    if ispriv then
        (r, w, xn) = (priv_r, priv_w, priv_xn);
    else
        (r, w, xn) = (user_r, user_w, user_xn);
else
    // Access from EL2 or EL3
    r = TRUE;
    w = perms.ap<2> == '0';
    xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elseif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW } then
        fail = !r || !w;
        failedread = !r;
    elseif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(48) UNKNOWN;
        return AArch64.PermissionFault(ipaddress, level, acctype,

```



```

                                !failedread, secondstage, s2fs1walk);
else
    return AArch64.NoFault();

```

aarch64/translation/checks/AArch64.CheckS2Permission

```

// AArch64.CheckS2Permission()
// =====
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(48) ipaddress,
                                       integer level, AccType acctype, boolean iswrite,
                                       boolean s2fs1walk, boolean hwupdatewalk)

    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    xn = perms.xn == '1';

    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
    elseif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elseif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    elseif hwupdatewalk then
        fail = !w;
        failedread = !iswrite;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();

```

aarch64/translation/debug/AArch64.CheckBreakpoint

```

// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);

```

```

elseif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    return AArch64.DebugFault(acctype, iswrite);
else
    return AArch64.NoFault();

```

aarch64/translation/debug/AArch64.CheckDebug

```

// AArch64.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, size);

    return fault;

```

aarch64/translation/debug/AArch64.CheckWatchpoint

```

// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elseif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();

```

aarch64/translation/faults/AArch64.AccessFlagFault

```

// AArch64.AccessFlagFault()
// =====

FaultRecord AArch64.AccessFlagFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);

```

aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// =====

FaultRecord AArch64.AddressSizeFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// =====

FaultRecord AArch64.AlignmentFault(ArchType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// =====
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(type, ipaddress, level, acctype, iswrite, extflag,
                                     secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// =====

FaultRecord AArch64.DebugFault(ArchType acctype, boolean iswrite)

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =====

FaultRecord AArch64.NoFault()

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =====

FaultRecord AArch64.PermissionFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// =====

FaultRecord AArch64.TranslationFault(bits(48) ipaddress, integer level,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress, level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

aarch64/translation/translation/AArch64.CheckAndUpdateDescriptor

```
// AArch64.CheckAndUpdateDescriptor()
// =====
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
                                             boolean secondstage, bits(64) vaddress, AccType acctype,
                                             boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk)

    // Check if access flag can be updated
    if result.AF && acctype != AccType_AT then
        if fault.type == Fault_None then
            hw_update_AF = TRUE;
        elseif ConstrainUnpredictable() == Constraint_TRUE then
            hw_update_AF = TRUE;
        else
            hw_update_AF = FALSE;

    // AP[2] / S2AP[2] is not updated for speculative access
    if result.AP && fault.type == Fault_None then
        write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW}) && !s2fs1walk;
```

```

hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC})) || hwupdatewalk;

if hw_update_AF || hw_update_AP then
    if secondstage || !HasS2Translation() then
        descaddr2 = result.descaddr;
    else
        hwupdatewalk = TRUE;
        descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8,
hwupdatewalk);
        if IsFault(descaddr2) then
            return descaddr2.fault;

    desc = _Mem[descaddr2, 8, AccType_ATOMICRW];

    if hw_update_AF then
        desc<10> = '1';
    if hw_update_AP then
        desc<7> = (if secondstage then '1' else '0');

    _Mem[descaddr2, 8, AccType_ATOMICRW] = desc;

return fault;

```

aarch64/translation/translation/AArch64.FirstStageTranslate

```

// AArch64.FirstStageTranslate()
// =====
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
boolean wasaligned, integer size)

    if HasS2Translation() then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[.].M == '1';

    ipaddress = bits(48) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS10ff(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
acctype != AccType_IFETCH) then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
S1.addrdesc.address.NS,
acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
acctype, iswrite,
secondstage, s2fs1walk);

```

```
// Check and update translation table descriptor if required
hwupdatewalk = FALSE;
s2fs1walk = FALSE;
S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault,
                                                    secondstage, vaddress, acctype,
                                                    iswrite, s2fs1walk, hwupdatewalk);

return S1.addrdesc;
```

aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =====
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                       boolean wasaligned, integer size)

// First Stage Translation
S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
if !IsFault(S1) && HasS2Translation() then
    s2fs1walk = FALSE;
    hwupdatewalk = FALSE;
    result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                         size, hwupdatewalk);
else
    result = S1;

return result;
```

aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// =====
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                              AccType acctype, boolean iswrite, boolean wasaligned,
                                              boolean s2fs1walk, integer size, boolean hwupdatewalk)

assert HasS2Translation();

s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
secondstage = TRUE;

if s2_enabled then // Second stage enabled
    ipaddress = S1.paddress.physicaladdress<47:0>;

    S2 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                       s2fs1walk, size);

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
        acctype != AccType_IFETCH) then
        S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

    if !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite, s2fs1walk, hwupdatewalk);

    // Check for instruction fetches from Device memory not marked as execute-never. As there
    // has not been a Permission Fault then the memory is not marked execute-never.
    if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
```

```

acctype, iswrite,
secondstage, s2fs1walk);

// Check for protected table walk
if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
    S2.addrdesc.memattrs.type == MemType_Device) then
    S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S2.level, acctype,
                                                iswrite, secondstage, s2fs1walk);

// Check and update translation table descriptor if required
S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                    secondstage, vaddress, acctype,
                                                    iswrite, s2fs1walk, hwupdatewalk);

result = CombineS1S2Desc(S1, S2.addrdesc);
else
    result = S1;

return result;

```

aarch64/translation/translation/AArch64.SecondStageWalk

```

// AArch64.SecondStageWalk()
// =====
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size, boolean hwupdatewalk)

assert HasS2Translation();

s2fs1walk = TRUE;
wasaligned = TRUE;
return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                    size, hwupdatewalk);

```

aarch64/translation/translation/AArch64.TranslateAddress

```

// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                          boolean wasaligned, integer size)

result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
    result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(vaddress);

return result;

```

aarch64/translation/walk/AArch64.TranslationTableWalk

```

// AArch64.TranslationTableWalk()
// =====
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(48) ipaddress, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,

```

```

                                boolean s2fs1walk, integer size)
if !secondstage then
    assert !ELUsingAArch32(S1TranslationRegime());
else
    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

TLBRecord result;
AddressDescriptor descaddr;
bits(64) baseregister;
bits(64) inputaddr;          // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

descaddr.memattrs.type = MemType_Normal;

// Derived parameters for the page table walk:
// grainsize = Log2(Size of Table)          - Size of Table is 4KB, 16KB or 64KB in AArch64
// stride = Log2(Address per Level)         - Bits of address consumed at each level
// firstblocklevel = First level where a block entry is allowed
// ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTOR_EL2.PS
// inputsize = Log2(Size of Input Address) - Input Address size in bits
// level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    top = AddrTop(inputaddr, PSTATE.EL);

    if PSTATE.EL == EL3 then
        inputsize = 64 - UInt(TCR_EL3.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL3.TG0 == '01';
        midgrain = TCR_EL3.TG0 == '10';
        ps = TCR_EL3.PS;
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
        disabled = FALSE;
        baseregister = TTBR0_EL3;
        descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGNO, TCR_EL3.IRGNO);
        reversedescriptors = SCTLR_EL3.EE == '1';
        lookupsecure = TRUE;
        singlepriv = TRUE;
        ha = TCR_EL3.HA;
        hd = TCR_EL3.HD;
        hierattrsdissabled = HaveHPDExt() && TCR_EL3.HPD == '1';
    elseif IsInHost() then
        if inputaddr<top> == '0' then
            inputsize = 64 - UInt(TCR_EL2.T0SZ);
            if inputsize > 48 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = 48;
            if inputsize < 25 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = 25;
            largegrain = TCR_EL2.TG0 == '01';
            midgrain = TCR_EL2.TG0 == '10';
            basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
            disabled = TCR_EL2.EPD0 == '1';
            baseregister = TTBR0_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGNO, TCR_EL2.IRGNO);
            hierattrsdissabled = HaveHPDExt() && TCR_EL2.HPD0 == '1';

```



```

else
    inputsize = 64 - UInt(TCR_EL2.T1SZ);
    if inputsize > 48 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 48;
    if inputsize < 25 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
        midgrain = TCR_EL2.TG1 == '01';
        basefound = inputsize >= 25 && inputsize <= 48 && IsOnes(inputaddr<top:inputsize>);
        disabled = TCR_EL2.EPD1 == '1';
        baseregister = TTBR1_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORG1, TCR_EL2.IRG1);
        hierattrsdissabled = HaveHPDExt() && TCR_EL2.HPD1 == '1';
    ps = TCR_EL2.IPS;
    reversedescriptors = SCTLRL_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = FALSE;
    ha = TCR_EL2.HA;
    hd = TCR_EL2.HD;
elseif PSTATE.EL == EL2 then
    inputsize = 64 - UInt(TCR_EL2.T0SZ);
    if inputsize > 48 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 48;
    if inputsize < 25 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL2.TG0 == '01';
        midgrain = TCR_EL2.TG0 == '10';
        ps = TCR_EL2.PS;
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
        disabled = FALSE;
        baseregister = TTBR0_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORG0, TCR_EL2.IRG0);
        reversedescriptors = SCTLRL_EL2.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;
        ha = TCR_EL2.HA;
        hd = TCR_EL2.HD;
        hierattrsdissabled = HaveHPDExt() && TCR_EL2.HPD == '1';
else
    if inputaddr<top> == '0' then
        inputsize = 64 - UInt(TCR_EL1.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
            largegrain = TCR_EL1.TG0 == '01';
            midgrain = TCR_EL1.TG0 == '10';
            basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
            disabled = TCR_EL1.EPD0 == '1';
            baseregister = TTBR0_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORG0, TCR_EL1.IRG0);
            hierattrsdissabled = HaveHPDExt() && TCR_EL1.HPD0 == '1';
    else
        inputsize = 64 - UInt(TCR_EL1.T1SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();

```

```

        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 48;
    if inputsize < 25 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL1.TG1 == '11';          // TG1 and TG0 encodings differ
        midgrain = TCR_EL1.TG1 == '01';
        basefound = inputsize >= 25 && inputsize <= 48 && IsOnes(inputaddr<top:inputsize>);
        disabled = TCR_EL1.EPD1 == '1';
        baseregister = TTBR1_EL1;
        descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORG1, TCR_EL1.IRG1);
        hierattrsdissabled = HaveHPDExt() && TCR_EL1.HPD1 == '1';
    ps = TCR_EL1.IPS;
    reversedescriptors = SCTLR_EL1.EE == '1';
    lookupsecure = IsSecure();
    singlepriv = FALSE;
    ha = TCR_EL1.HA;
    hd = TCR_EL1.HD;
    if largegrain then
        grainsize = 16;                                // Log2(64KB page size)
        firstblocklevel = 2;                            // Largest block is 512MB (2^29 bytes)
    elseif midgrain then
        grainsize = 14;                                // Log2(16KB page size)
        firstblocklevel = 2;                            // Largest block is 32MB (2^25 bytes)
    else // Small grain
        grainsize = 12;                                // Log2(4KB page size)
        firstblocklevel = 1;                            // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                        // Log2(page size / 8 bytes)
        // The starting level is the number of strides needed to consume the input address
        level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));
else
    // Second stage translation
    inputaddr = ZeroExtend(ipaddress);
    inputsize = 64 - UInt(VTCR_EL2.T0SZ);
    if inputsize > 48 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 48;
    if inputsize < 25 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 25;
    largegrain = VTCR_EL2.TG0 == '01';
    midgrain = VTCR_EL2.TG0 == '10';
    ps = VTCR_EL2.PS;
    basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<63:inputsize>);
    disabled = FALSE;
    baseregister = VTTBR_EL2;
    descaddr.memattrs = WalkAttrDecode(VTCR_EL2.IRG0, VTCR_EL2.ORG0, VTCR_EL2.SH0);
    reversedescriptors = SCTLR_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;
    ha = VTCR_EL2.HA;
    hd = VTCR_EL2.HD;

    startlevel = UInt(VTCR_EL2.SL0);
    if largegrain then
        grainsize = 16;                                // Log2(64KB page size)
        level = 3 - startlevel;
        firstblocklevel = 2;                            // Largest block is 512MB (2^29 bytes)
    elseif midgrain then
        grainsize = 14;                                // Log2(16KB page size)
        level = 3 - startlevel;
        firstblocklevel = 2;                            // Largest block is 32MB (2^25 bytes)
    else // Small grain
        grainsize = 12;                                // Log2(4KB page size)

```

```

    level = 2 - startlevel;
    firstblocklevel = 1; // Largest block is 1GB (2^30 bytes)
    stride = grainsize - 3; // Log2(page size / 8 bytes)

    // Limits on IPA controls based on implemented PA size. Level 0 is only
    // supported by small grain translations
    if largegrain then // 64KB pages
        // Level 1 only supported if implemented PA size is greater than 2^42 bytes
        if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
    elseif midgrain then // 16KB pages
        // Level 1 only supported if implemented PA size is greater than 2^40 bytes
        if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
    else // Small grain, 4KB pages
        // Level 0 only supported if implemented PA size is greater than 2^42 bytes
        if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

    // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
    inputsizecheck = inputsize;
    if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
        case ConstrainUnpredictable() of
            when Constraint_FORCE
                // Restrict the inputsize to the PAMax value
                inputsize = PAMax();
                inputsizecheck = PAMax();
            when Constraint_FORCENOSLCHECK
                // As FORCE, except use the configured inputsize in the size checks below
                inputsize = PAMax();
            when Constraint_FAULT
                // Generate a translation fault
                basefound = FALSE;
            otherwise
                Unreachable();

    // Number of entries in the starting level table =
    // (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is: startsizecheck < Log2(2 entries)
    // Upper bound check is: startsizecheck > Log2(pagesize/8*16)
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

    if !basefound || disabled then
        level = 0; // AArch32 reports this as a level 1 fault
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype, iswrite,
            secondstage, s2fs1walk);
        return result;

    case ps of
        when '000' outputsize = 32;
        when '001' outputsize = 36;
        when '010' outputsize = 40;
        when '011' outputsize = 42;
        when '100' outputsize = 44;
        when '101' outputsize = 48;
        otherwise outputsize = 48;

    if outputsize > PAMax() then outputsize = PAMax();

    if outputsize != 48 && !IsZero(baseregister<47:outputsize>) then
        level = 0;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype, iswrite,
            secondstage, s2fs1walk);
        return result;

    // Bottom bound of the Base address is:
    // Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
    // Number of entries in starting level table =

```

```
// (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize); // Log2(Num of entries*8)
baseaddress = baseregister<47:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(48) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.physicaladdress = baseaddress OR index;
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if secondstage || !HasS2Translation() then
        descaddr2 = descaddr;
    else
        hwupdatewalk = FALSE;
        descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    desc = _Mem[descaddr2, 8, AccType_PTW];
    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
        // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
            iswrite, secondstage, s2fslwalk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then // Block (01) or Page (11)
        blocktranslate = TRUE;
    else // Table (11)
        if outputsize != 48 && !IsZero(desc<47:outputsize>) then
            result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                iswrite, secondstage, s2fslwalk);
            return result;

    baseaddress = desc<47:grainsize>:Zeros(grainsize);

    if !secondstage then
        // Unpack the upper and lower table attributes
        ns_table = ns_table OR desc<63>;
    if !secondstage && !hierattrsdissabled then
        ap_table<1> = ap_table<1> OR desc<62>; // read-only
        xn_table = xn_table OR desc<60>;
        // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
        if !singlepriv then
            ap_table<0> = ap_table<0> OR desc<61>; // privileged
            pxn_table = pxn_table OR desc<59>;

    level = level + 1;
    addrselecttop = addrselectbottom - 1;
    blocktranslate = FALSE;
until blocktranslate;
```

```
// Check block size is supported at this level
if level < firstblocklevel then
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
    return result;

// Check for misprogramming of the contiguous bit
if largegrain then
    contiguousbitcheck = level == 2 && inputsizesize < 34;
elseif midgrain then
    contiguousbitcheck = level == 2 && inputsizesize < 30;
else
    contiguousbitcheck = level == 1 && inputsizesize < 34;

if contiguousbitcheck && desc<52> == '1' then
    if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
        return result;

// Check the output address is inside the supported range
if outputsizesize != 48 && !IsZero(desc<47:outputsizesize>) then
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;

// Check for hardware update of AF and AP[2]/S2AP[2]
hwupdate_access_flag = (HaveAccessFlagUpdateExt() && ha == '1');
hwupdate_access_permission = (HaveDirtyBitMechanismExt() && hwupdate_access_flag && hd == '1');

// Check Access Flag
if desc<10> == '0' then
    if !hwupdate_access_flag then
        result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
        return result;
    else
        result.descupdate.AF = TRUE;

if hwupdate_access_permission && desc<51> == '1' then
    // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
    if !secondstage && desc<7> == '1' then
        desc<7> = '0';
        result.descupdate.AP = TRUE;
    elseif secondstage && desc<7> == '0' then
        desc<7> = '1';
        result.descupdate.AP = TRUE;

// Required descriptor if AF or AP[2]/S2AP[2] needs update
result.descupdate.descaddr = descaddr;

xn = desc<54>;
pxn = desc<53>;
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
ap = desc<7:6>:'1';
memattr = desc<5:2>; // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN; // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
```

```

if !secondstage then
    result.perms.xn      = xn OR xn_table;
    result.perms.ap<2>  = ap<2> OR ap_table<1>;          // Force read-only
    // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
        result.perms.pxn   = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn   = '0';
        result.nG          = '0';
        result.perms.ap<0> = '1';
        result.addrdesc.memattr = S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>  = '1';
        result.perms.xn      = xn;
        result.perms.pxn     = '0';
        result.nG           = '0';
        result.addrdesc.memattr = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

result.addrdesc.paddress.physicaladdress = outputaddress;
result.addrdesc.fault = AArch64.NoFault();
result.contiguous = contiguousbit == '1';

return result;

```

E1.3 Library pseudocode for AArch32

E1.3.1 aarch32/debug

aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word<UInt(vaddress<4:2>) + 24> = '1'; // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1'; // Secure vectors (or no EL3)
        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1'; // Monitor vectors

        // Mask out bits not corresponding to vectors.
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        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();
        else
            match = FALSE;

    return match;
```

aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // In the recommended interface, SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.
```

```
(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.BRPs);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool();

    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();
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool();

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```

aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

```
// AArch32.BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(DBGDIDR.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR[n].BT;
```



```

if ((type IN {'011x', '11xx'}) && !HaveVirtHostExt()) ||           // Context matching
    (type == '010x' && HaltOnBreakpointOrWatchpoint()) ||         // Address mismatch
    (type != '0x0x' && !context_aware) ||                         // Context matching
    (type == '1xxx' && !HaveEL(EL2))) then                         // EL2 extension
    (c, type) = ConstrainUnpredictableBits();
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE, FALSE);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (type == '0x0x');
mismatch   = (type == '010x');
match_vmid = (type == '10xx');
match_cid1 = (type == 'xx1x');
match_cid2 = (type == '11xx');
linked     = (type == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return (FALSE, FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE, FALSE);

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    assert byte IN {0, 2}; // "vaddress" is halfword aligned.
    byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    BVR_match = vaddress<31:2> == DBGXVR[n]<31:2> && byte_select_match;
elseif match_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGXVR[n]<31:0>);

if match_vmid then
    if ELUsingAArch32(EL2) then
        vmid = ZeroExtend(VTTBR.VMID, 16);
        bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
    elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGXVR[n]<15:0>;
    BVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        vmid == bvr_vmid);
elseif match_cid2 then
    BVR_match = (!IsSecure() && HaveVirtHostExt() &&
        !ELUsingAArch32(EL2) &&
        DBGXVR[n]<31:0> == CONTEXTIDR_EL2);

bvr_match_valid = (match_addr || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BVR_match) && (!bvr_match_valid || BVR_match);

return (match && !mismatch, !match && mismatch);

```

aarch32/debug/breakpoint/AArch32.StateMatch

```

// AArch32.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
    boolean isbreakpt, boolean ispriv)

```

```
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
if ((HMC:SSC:PxC) IN {'011xx', '100x0', '101x0', '11010', '11101', '1111x'}) || // Reserved
    (HMC == '0' && PxC == '00' && !isbreakpt) || // Usr/Svc/Sys
    (SSC IN {'01', '10'} && !HaveEL(EL3)) || // No EL3
    (HMC:SSC:PxC == '11000' && ELUsingAArch32(EL3)) || // AArch64 only
    (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL(EL3) && !HaveEL(EL2)) || // No EL3/EL2
    (HMC:SSC:PxC == '11100' && !HaveEL(EL2))) then // No EL2
    (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

PL2_match = HaveEL(EL2) && HMC == '1';
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpt && HMC == '0' && PxC == '00' && SSC != '11';

if SSU_match then
    priv_match = PSTATE.M IN {M32_User, M32_Svc, M32_System};
else
    case PSTATE.EL of
        when EL3, EL1 priv_match = if ispriv then PL1_match else PL0_match;
        when EL2 priv_match = PL2_match;
        when EL0 priv_match = PL0_match;

    case SSC of
        when '00' security_state_match = TRUE; // Both
        when '01' security_state_match = !IsSecure(); // Non-secure only
        when '10' security_state_match = IsSecure(); // Secure only
        when '11' security_state_match = TRUE; // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPS));
        last_ctx_cmp = UInt(DBGDIDR.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED return FALSE; // Disabled
                when Constraint_NONE linked = FALSE; // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
        (linked_match, -) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN; // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        spd = (if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32);
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```

aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()

    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();

    pmuirq = (PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1');
    for n = 0 to UInt(PMCR.N) - 1
        if HaveEL(EL2) then
            hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UInt(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

aarch32/debug/pmu/AArch32.CountEvents

```
// AArch32.CountEvents()
// =====
// Return TRUE if counter "n" should count its event.

boolean AArch32.CountEvents(integer n)
    assert(n == 31 || n < UInt(PMCR.N));

    if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);
```

```
// Event counting is disabled in Debug state
debug = Halted();

if HaveEL(EL2) then
    hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
    hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    E = (if n < UInt(hpmn) || n == 31 then PMCR.E else hpme);
else
    E = PMCR.E;
enabled = (E == '1' && PMCNTENSET<n> == '1');

// Event counting might be prohibited
prohibited = AArch32.ProfilingProhibited(IsSecure(), PSTATE.EL);
if PSTATE.EL == EL2 && HaveHPMDExt() && (n < UInt(hpmn) || n == 31) then
    hpmn = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
    hpme = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME);
    prohibited = (hpmn == '1' && !ExternalSecureNoninvasiveDebugEnabled());
if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = (if n == 31 then PMCCFILTR<31:27> else PMEVTYPEPER[n]<31:27>);

H = if !HaveEL(EL2) then '0' else filter<0>;
P = filter<4>; U = filter<3>;
if !IsSecure() && HaveEL(EL3) then
    P = P EOR filter<2>; U = U EOR filter<1>;

case PSTATE.EL of
    when EL0    filtered = U == '1';
    when EL1,EL3 filtered = P == '1';
    when EL2    filtered = H == '0';

return !debug && enabled && !prohibited && !filtered;
```

aarch32/debug/pmu/AArch32.ProfilingProhibited

```
// AArch32.ProfilingProhibited()
// =====
// Determine whether event counting is prohibited in the current state.

boolean AArch32.ProfilingProhibited(boolean secure, bits(2) el)

    if (el == EL0 && !ELUsingAArch32(EL1)) || !ELUsingAArch32(el) then
        return AArch64.ProfilingProhibited(secure, el);

    // Events are always counted in Non-secure state.
    if !secure then return FALSE;

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    if !HaveEL(EL3) then return FALSE;

    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    // * EL3 is using AArch32 and SDCR.SPME == 1
    spme = (if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME);
    if spme == '1' then return FALSE;

    // * Allowed by the IMPLEMENTATION DEFINED authentication interface
    if ExternalSecureNoninvasiveDebugEnabled() then return FALSE;

    // * EL3 or EL1 is using AArch32, executing at EL0, and SDER.SUNIDEN == 1.
    if el == EL0 && ELUsingAArch32(EL1) && SDER.SUNIDEN == '1' then return FALSE;

    return TRUE;
```

aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFIELDS();
    EndOfInstruction();
```

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)
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFIELDS(); // Update EDSCR processor state flags.
    EndOfInstruction();
```

aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
```

```
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
PSTATE.T = '1'; // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DPSR = bits(32) UNKNOWN;
PSTATE.E = SCTLRE.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
if HavePANExt() then
    if !from_secure then
        PSTATE.PAN = '0';
    elseif SCTLRE.SPAN == '0' then
        PSTATE.PAN = '1';
EDSCR.ERR = '1';
UpdateEDSCRFields(); // Update EDSCR processor state flags.
EndOfInstruction();
```

aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    bottom = if DBGWVR[n]<2> == '1' then 2 else 3; // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '1111111', 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();
    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();
            bottom = 3; // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE; // Disabled
            when Constraint_NONE mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<31:mask> == DBGWVR[n]<31:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;

    return WVR_match && byte_select_match;
```

aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
```

```

        boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.WRPs);

    // "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
        linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;

```

E1.3.2 aarch32/exceptions

aarch32/exceptions/aborts/AArch32.Abort

```

// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elseif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);

```

aarch32/exceptions/aborts/AArch32.AbortSyndrome

```

// AArch32.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception type, FaultRecord fault, bits(32) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type == Exception_DataAbort;

    exception.syndrome = FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);

```

```

if IPValid(fault) then
    exception.ipvalid = TRUE;
    exception.ipaddress = ZeroExtend(fault.ipaddress);
else
    exception.ipvalid = FALSE;

return exception;

```

aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```

// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()

bits(32) pc = ThisInstrAddr();
if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
    if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

    // Generate an Alignment fault Prefetch Abort exception
    vaddress = pc;
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    secondstage = FALSE;
    AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));

```

aarch32/exceptions/aborts/AArch32.ReportDataAbort

```

// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

// The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
// format. Normally, the current translation table format determines the format. For an abort
// from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
// any of the following applies:
// * The Secure TTBCR.EAE is set to 1.
// * The abort is synchronous and either:
//   - It is taken from Hyp mode.
//   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
long_format = FALSE;
if route_to_monitor && !IsSecure() then
    long_format = TTBCR.S.EAE == '1';
    if !IsAsyncAbort(fault) && !long_format then
        long_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
else
    long_format = TTBCR.EAE == '1';

d_side = TRUE;
if long_format then
    syndrome = AArch32.FaultStatusLD(d_side, fault);
else
    syndrome = AArch32.FaultStatusSD(d_side, fault);

if fault.acctype == AccType_IC then
    if (!long_format &&
        boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
        i_syndrome = syndrome;
        syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
    else
        i_syndrome = bits(32) UNKNOWN;
    if route_to_monitor then
        IFSR_S = i_syndrome;
    else

```



```

        IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;

```

aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```

// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;

```

aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```

// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' || IsSecondStage(fault) ||
        (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRExt.MOE = fault.debugmoe;

    if route_to_monitor then

```

```

    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
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);

```

aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```

// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure()) && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' || IsSecondStage(fault) ||
         (IsDebugException(fault) && HDCR.TDE == '1'));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRExt.MOE = fault.debugmoe;

    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        if fault.type == Fault_Alignment then // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAlignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

```

aarch32/exceptions/asynch/AArch32.TakePhysicalAsynchAbortException

```

// AArch32.TakePhysicalAsynchAbortException()
// =====

AArch32.TakePhysicalAsynchAbortException(boolean parity, bit extflag,
                                         boolean syndrome_valid, bits(24) full_syndrome)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSystemErrorException(syndrome_valid, full_syndrome);

```

```

route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
    (HCR.TGE == '1' || HCR.AMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;

fault = AArch32.AsynchExternalAbort(parity, extflag);
vaddress = bits(32) UNKNOWN;

if route_to_monitor then
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
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);

```

aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException

```

// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.FIQ == '1';

if route_to_aarch64 then AArch64.TakePhysicalFIQException();

route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
    (HCR.TGE == '1' || HCR.FMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;

if route_to_monitor then
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
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);

```

aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException

```

// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then

```

```

    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
        (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif 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);

```

aarch32/exceptions/asynch/AArch32.TakeVirtualAsynchAbortException

```

// AArch32.TakeVirtualAsynchAbortException()
// =====

AArch32.TakeVirtualAsynchAbortException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then // Virtual Asynchronous Abort enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSystemErrorException();

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    extflag = bit IMPLEMENTATION_DEFINED "Virtual Asynchronous Abort ExT bit";
    fault = AArch32.AsynchExternalAbort(parity, extflag);

    if ELUsingAArch32(EL2) then HCR.VA = '0'; else HCR_EL2.VSE = '0';

    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

```

aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

```

// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

```

```
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;

AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// =====

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH; // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

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';
```

aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Return the Exception Class and Instruction Length fields for reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception type)

    il = if ThisInstrLength() == 32 then '1' else '0';

    case type of
        when Exception_Uncategorized    ec = 0x00; il = '1';
```

```

when Exception_WFxTrap          ec = 0x01;
when Exception_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_CP14RRTTrap      ec = 0x0C;
when Exception_IllegalState     ec = 0x0E; il = '1';
when Exception_SupervisorCall   ec = 0x11;
when Exception_HypervisorCall   ec = 0x12;
when Exception_MonitorCall      ec = 0x13;
when Exception_InstructionAbort  ec = 0x20; il = '1';
when Exception_PCAlignment      ec = 0x22; il = '1';
when Exception_DataAbort        ec = 0x24;
when Exception_FPtrappedException ec = 0x28;
otherwise                       Unreachable();

if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
    ec = ec + 1;

return (ec,il);

```

aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```

// AArch32.GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));

```

aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```

// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception type = exception.type;

    (ec,il) = AArch32.ExceptionClass(type);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif type == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else

```

```
HPFAR<31:4> = bits(28) UNKNOWN;

return;
```

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);
```

aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0'; // Secure state
    elseif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLAR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    PSTATE.T = SCTLAR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLAR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType_UNKNOWN);
```

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;
```

aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, element, accumulated_exceptions);

    FPExc.DEX = '1';
    FPExc.TFV = '1';
    FPExc<7,4:0> = accumulated_exceptions<7,4:0>; // IDF, IXF, UFF, OFF, DZF, IOF

    AArch32.TakeUndefInstrException();
```

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);
```

aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;

    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCEException(immediate);
```

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);
```

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);
```

aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// =====

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();

    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

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)
```

```

assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

spsr = GetPSRFromPSTATE();
if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
    AArch32.ReportHypEntry(exception);
AArch32.WriteMode(M32_Hyp);
SPSR[] = spsr;
ELR_hyp = preferred_exception_return;
PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
PSTATE.SS = '0';
if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
PSTATE.E = HSCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
BranchTo(HVBAR + vect_offset, BranchType_UNKNOWN);
EndOfInstruction();

```

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)
assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

spsr = GetPSRFromPSTATE();
if PSTATE.M == M32_Monitor then SCR.NS = '0';
AArch32.WriteMode(target_mode);
SPSR[] = spsr;
R[14] = preferred_exception_return + lr_offset;
PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
PSTATE.SS = '0';
if target_mode == M32_FIQ then
    PSTATE.<A,I,F> = '111';
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';
BranchTo(ExcVectorBase() + vect_offset, BranchType_UNKNOWN);
EndOfInstruction();

```

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)
assert HaveEL(EL3) && ELUsingAArch32(EL3);
from_secure = IsSecure();
spsr = GetPSRFromPSTATE();
if PSTATE.M == M32_Monitor then SCR.NS = '0';
AArch32.WriteMode(M32_Monitor);
SPSR[] = spsr;
R[14] = preferred_exception_return + lr_offset;
PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
PSTATE.SS = '0';

```

```
PSTATE.<A,I,F> = '111';
PSTATE.E = SCTL.R.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
if HavePANExt() then
    if !from_secure then
        PSTATE.PAN = '0';
    elseif SCTL.R.SPAN == '0' then
        PSTATE.PAN = '1';
BranchTo(MVBAR + vect_offset, BranchType_UNKNOWN);
EndOfInstruction();
```

aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrap

```
// AArch32.AArch32SystemAccessTrap()
// =====
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch32.AArch32SystemAccessTrap(bits(2) target_el, bits(32) instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if !ELUsingAArch32(target_el) || AArch32.GeneralExceptionsToAArch64() then
        AArch64.AArch32SystemAccessTrap(target_el, instr);

    assert target_el IN {EL1,EL2};

    if target_el == EL2 then
        exception = AArch32.AArch32SystemAccessTrapSyndrome(instr);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrapSyndrome

```
// AArch32.AArch32SystemAccessTrapSyndrome()
// =====
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.AArch32SystemAccessTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10 exception = ExceptionSyndrome(Exception_FPIDTrap);
            when 14 exception = ExceptionSyndrome(Exception_CP14RTTTrap);
            when 15 exception = ExceptionSyndrome(Exception_CP15RTTTrap);
            otherwise Unreachable();
        iss<19:17> = instr<7:5>; // opc2
        iss<16:14> = instr<23:21>; // opc1
        iss<13:10> = instr<19:16>; // CRn
        iss<8:5> = instr<15:12>; // Rt
        iss<4:1> = instr<3:0>; // CRm
    elseif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14 exception = ExceptionSyndrome(Exception_CP14RRTTTrap);
            when 15 exception = ExceptionSyndrome(Exception_CP15RRTTTrap);
            otherwise Unreachable();
        iss<19:16> = instr<7:4>; // opc1
        iss<13:10> = instr<19:16>; // Rt2
        iss<8:5> = instr<15:12>; // Rt
```

```

    iss<4:1> = instr<3:0>; // CRm
elseif instr<27:25> == '110' && instr<31:28> != '1111' then
    // LDC/STC
    assert cpnum == 14;
    exception = ExceptionSyndrome(Exception_CP14DITrap);
    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 // Literal addressing
        iss<8:5> = bits(4) UNKNOWN;
        iss<3> = '1';
    else
        iss<8:5> = instr<19:16>; // Rn
        iss<3> = '0';
    else
        Unreachable();
    iss<0> = instr<20>; // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0> = iss;

    return exception;

```

aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEEnabled

```

// AArch32.CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEEnabled(boolean fpexc_check, boolean advsimd)

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        AArch64.CheckFPAAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            // Check if access disabled in CPACR
            case cpacr_cp10 of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '11' disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAAdvSIMDTrap(advsimd); // Also check against HCPTR and CPTR_EL3

```

aarch32/exceptions/traps/AArch32.CheckFPAAdvSIMDTrap

```

// AArch32.CheckFPAAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAAdvSIMDTrap(boolean advsimd)

```

```

if HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
    AArch64.CheckFPAdvSIMDTrap();
else
    if HaveEL(EL2) && !IsSecure() then
        hcptr_tase = HCPTR.TASE;
        hcptr_cp10 = HCPTR.TCP10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
            if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                exception.syndrome<24:20> = ConditionSyndrome();

            if advsimd then
                exception.syndrome<5> = '1';
            else
                exception.syndrome<5> = '0';
                exception.syndrome<3:0> = '1010'; // coproc field, always 0xA

            if PSTATE.EL == EL2 then
                AArch32.TakeUndefInstrException(exception);
            else
                AArch32.TakeHypTrapException(exception);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

return;

```

aarch32/exceptions/traps/AArch32.CheckForSMCTrap

```

// AArch32.CheckForSMCTrap()
// =====
// Check for trap on SMC instruction

AArch32.CheckForSMCTrap()

if HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
    AArch64.CheckForSMCTrap(Zeros(16));
else
    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && HCR.TSC == '1';
    if route_to_hyp then
        exception = ExceptionSyndrome(Exception_MonitorCall);
        AArch32.TakeHypTrapException(exception);

```

aarch32/exceptions/traps/AArch32.CheckForWFXTrap

```

// AArch32.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFXTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFXTrap(target_el, is_wfe);
        return;

    case target_el of
        when EL1 trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';

```

```

when EL2 trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
when EL3 trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

if trap then
    if (target_el == EL1 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) &&
        HCR_EL2.TGE == '1') then
        AArch64.WFxTrap(target_el, is_wfe);

    if target_el == EL3 then
        AArch32.TakeMonitorTrapException();
    elseif target_el == EL2 then
        exception = ExceptionSyndrome(Exception_WFxTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        exception.syndrome<0> = if is_wfe then '1' else '0';
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();

```

aarch32/exceptions/traps/AArch32.CheckITEnabled

```

// AArch32.CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)

    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[.].ITD);

    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
    next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
        '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxx111'} 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;

```

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();
    elseif PSTATE.IL == '1' then
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then

```

```

        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.TakeUndefInstrException();

```

aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```

// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

```

```

AArch32.CheckSETENDEnabled()

    if PSTATE.EL == EL2 then
        setend_disabled = HCTL.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTL.SED else SCTL[.].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;

```

aarch32/exceptions/traps/AArch32.TakeHypTrapException

```

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

```

```

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);

```

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);

```

aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```

// AArch32.TakeUndefInstrException()
// =====

```

```

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

```

```

// AArch32.TakeUndefInstrException()
// =====

```

```

AArch32.TakeUndefInstrException(ExceptionRecord exception)

```

```

route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x04;
lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
elseif route_to_hyp then
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);

```

aarch32/exceptions/traps/AArch32.UndefinedFault

```

// AArch32.UndefinedFault()
// =====

AArch32.UndefinedFault()

if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();

AArch32.TakeUndefInstrException();

```

E1.3.3 aarch32/functions

aarch32/functions/aborts/AArch32.CreateFaultRecord

```

// AArch32.CreateFaultRecord()
// =====

FaultRecord AArch32.CreateFaultRecord(Fault type, bits(40) ipaddress, bits(4) domain,
                                       integer level, AccType acctype, boolean write, bit extflag,
                                       bits(4) debugmoe, boolean secondstage, boolean s2fs1walk)

FaultRecord fault;
fault.type = type;
if (type != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk &&
    AArch32.DomainValid(type, level)) then
    fault.domain = domain;
else
    fault.domain = bits(4) UNKNOWN;
fault.debugmoe = debugmoe;
fault.ipaddress = ZeroExtend(ipaddress);
fault.level = level;
fault.acctype = acctype;
fault.write = write;
fault.extflag = extflag;
fault.secondstage = secondstage;
fault.s2fs1walk = s2fs1walk;

return fault;

```

aarch32/functions/aborts/AArch32.DomainValid

```

// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault type, integer level)
    assert type != Fault_None;

```



```

case type of
  when Fault_Domain
    return TRUE;
  when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
    return level == 2;
  otherwise
    return FALSE;

```

aarch32/functions/aborts/AArch32.FaultStatusLD

```

// AArch32.FaultStatusLD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
  assert fault.type != Fault_None;

  bits(32) fsr = Zeros();
  if d_side then
    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
      fsr<13> = '1'; fsr<11> = '1';
    else
      fsr<11> = if fault.write then '1' else '0';
  if IsExternalAbort(fault) then fsr<12> = fault.extflag;
  fsr<9> = '1';
  fsr<5:0> = EncodeLDFSC(fault.type, fault.level);

  return fsr;

```

aarch32/functions/aborts/AArch32.FaultStatusSD

```

// AArch32.FaultStatusSD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
  assert fault.type != Fault_None;

  bits(32) fsr = Zeros();
  if d_side then
    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
      fsr<13> = '1'; fsr<11> = '1';
    else
      fsr<11> = if fault.write then '1' else '0';
  if IsExternalAbort(fault) then fsr<12> = fault.extflag;
  fsr<9> = '0';
  fsr<10,3:0> = EncodeSDFSC(fault.type, fault.level);
  if d_side then
    fsr<7:4> = fault.domain; // Domain field (data fault only)

  return fsr;

```

aarch32/functions/aborts/EncodeSDFSC

```

// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault type, integer level)

  bits(5) result;
  case type of
    when Fault_AccessFlag
      assert level IN {1,2};

```

```

    result = if level == 1 then '00011' else '00110';
when Fault_Alignment
    result = '00001';
when Fault_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
when Fault_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
when Fault_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
when Fault_SyncExternal
    result = '01000';
when Fault_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
when Fault_SyncParity
    result = '11001';
when Fault_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
when Fault_AsyncParity
    result = '11000';
when Fault_AsyncExternal
    result = '10110';
when Fault_Debug
    result = '00010';
when Fault_TLBConflict
    result = '10000';
when Fault_Lockdown
    result = '10100'; // IMPLEMENTATION DEFINED
when Fault_Exclusive
    result = '10101'; // IMPLEMENTATION DEFINED
when Fault_ICacheMaint
    result = '00100';
otherwise
    Unreachable();

return result;

```

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;

```

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);

```

aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL; shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX; shift_n = 1;
            else
                shift_t = SRType_ROR; shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00' shift_t = SRType_LSL;
        when '01' shift_t = SRType_LSR;
        when '10' shift_t = SRType_ASR;
        when '11' shift_t = SRType_ROR;
    return shift_t;
```

aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

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);
```

aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

aarch32/functions/common/Shift

```
// Shift()
// =====
```

```
bits(N) Shift(bits(N) value, SRTYPE type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;
```

aarch32/functions/common/Shift_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRTYPE type, integer amount, bit carry_in)
    assert !(type == SRTYPE_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRTYPE_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRTYPE_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRTYPE_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRTYPE_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRTYPE_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

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;
```

aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
             (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

aarch32/functions/coproc/AArch32.CheckSystemAccess

```
// AArch32.CheckSystemAccess()
// =====
// Check System register access instruction for enables and disables

AArch32.CheckSystemAccess(integer cp_num, bits(32) instr)
    assert cp_num == UInt(instr<11:8>) && (cp_num IN {14,15});

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        AArch64.CheckAArch32SystemAccess(instr);
        return;

    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then // MRC/MCR
        cpdt = TRUE; cpdt = FALSE; nreg = 1;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn = UInt(instr<19:16>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:21> == '1100010' then // MRRC/MCRR
        cpdt = TRUE; cpdt = FALSE; nreg = 2;
        opc1 = UInt(instr<7:4>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then // LDC/STC
        cpdt = FALSE; cpdt = TRUE; nreg = 0;
        opc1 = 0;
        CRn = UInt(instr<15:12>);
    else
        allocated = FALSE;

    //
    // Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
    // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
    if cp_num == 14 then
        // LDC and STC only supported for c5 in CP14 encoding space
        if cpdt && CRn != 5 then
            allocated = FALSE;
        else
            // Coarse-grained decode of CP14 based on opc1 field
```

```

        case opc1 of
            when 0      allocated = CP14DebugInstrDecode(instr);
            when 1      allocated = CP14TraceInstrDecode(instr);
            when 7      allocated = CP14JazelleInstrDecode(instr);    // JIDR only
            otherwise   allocated = FALSE;                          // All other values are unallocated

    elseif cp_num == 15 then
        // LDC and STC not supported in CP15 encoding space
        if !cprt then
            allocated = FALSE;
        else
            allocated = CP15InstrDecode(instr);

            // Coarse-grain traps to EL2 have a higher priority than Undefined Instruction
            if AArch32.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
                // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
                // Non-secure User mode is UNDEFINED when the trap is disabled, then it is
                // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
                // when the trap is enabled.
                if PSTATE.EL == EL0 && !IsSecure() && !allocated then
                    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
                        UNDEFINED;
                        AArch32.AArch32SystemAccessTrap(EL2, instr);

            else
                allocated = FALSE;

        if !allocated then
            UNDEFINED;

        // If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
        AArch32.CheckSystemAccessTraps(instr);

    return;

```

aarch32/functions/coproc/AArch32.CheckSystemAccessTraps

```

// Check for configurable disables or traps to a higher EL of an System register access.
AArch32.CheckSystemAccessTraps(bits(32) instr);

```

aarch32/functions/coproc/CP14DebugInstrDecode

```

// Decodes an accepted access to a debug System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14DebugInstrDecode(bits(32) instr);

```

aarch32/functions/coproc/CP14JazelleInstrDecode

```

// Decodes an accepted access to a Jazelle System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14JazelleInstrDecode(bits(32) instr);

```

aarch32/functions/coproc/CP14TraceInstrDecode

```

// Decodes an accepted access to a trace System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14TraceInstrDecode(bits(32) instr);

```

aarch32/functions/coproc/CP15InstrDecode

```

// Decodes an accepted access to a System register in the CP15 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP15InstrDecode(bits(32) instr);

```

aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// =====

// Return TRUE if the Exclusive 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 Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

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);
```

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);
```

aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// =====
```

```
// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.
```

```
AArch32.SetExclusiveMonitors(bits(32) address, integer size)
```

```
acctype = AccType_ATOMIC;
iswrite = FALSE;
aligned = (address != Align(address, size));

memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return;

if memaddrdesc.memattrs.shareable then
    MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

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;
```

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;
```

aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

CheckCryptoEnabled32()
CheckAdvSIMDEnabled();
// Return from CheckAdvSIMDEnabled() occurs only if access is permitted
return;
```


aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

CheckVFPEEnabled(boolean include_fpxc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDorFPEEnabled(include_fpxc_check, advsimd);
    // Return from CheckAdvSIMDorFPEEnabled() occurs only if VFP access is permitted
    return;
```

aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {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 == FType_Infinity); inf2 = (type2 == FType_Infinity);
        zero1 = (type1 == FType_Zero); zero2 = (type2 == FType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(32) FPRSqrtStep(bits(32) op1, bits(32) op2)
    FPCRType fpcr = StandardFPCRValue();
    (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 == FType_Infinity); inf2 = (type2 == FType_Infinity);
        zero1 = (type1 == FType_Zero); zero2 = (type2 == FType_Zero);
        bits(32) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
            result = FPHalvedSub(FPThree('0'), product, fpcr);
    return result;
```

aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

bits(32) FPRecipStep(bits(32) op1, bits(32) op2)
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FType_Infinity); inf2 = (type2 == FType_Infinity);
        zero1 = (type1 == FType_Zero); zero2 = (type2 == FType_Zero);
        bits(32) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
            result = FPSub(FPTwo('0'), product, fpcr);
    return result;
```

aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// =====

FPCRType StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '1100000000000000000000000000';
```

aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// =====

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                                boolean iswrite)

    if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
        A = SCTLR[]A; //use AArch64 register, when higher Exception level is using AArch64
    elseif PSTATE.EL == EL2 then
        A = HSCTLR.A;
    else
        A = SCTLR.A;
    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED };
    vector = acctype == AccType_VEC;

    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle(bits(32) address, integer size, AccType acctype, boolean wasaligned)
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);
```

```

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    value = _Mem[memaddrdesc, size, acctype];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    _Mem[memaddrdesc, size, acctype] = value;
    return;

```

aarch32/functions/memory/Hint_PreloadData

```
Hint_PreloadData(bits(32) address);
```

aarch32/functions/memory/Hint_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

aarch32/functions/memory/Hint_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```

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;
```

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;
```

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;
```

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;
```

aarch32/functions/memory/Mem_with_type

```
// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.
```

```

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;
    return;

```

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;

```

aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```

// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

return;

```

aarch32/functions/registers/AArch32.ResetSpecialRegisters

```

// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

return;

```

aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

aarch32/functions/registers/ALUExceptionReturn

```

// ALUExceptionReturn()
// =====

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elseif PSTATE.M IN {M32_User, M32_System} then

```

```

        UNPREDICTABLE;          // UNDEFINED or NOP
    else
        AArch32.ExceptionReturn(address, SPSR[]);

```

aarch32/functions/registers/ALUWritePC

```

// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address);
    else
        BranchWritePC(address);

```

aarch32/functions/registers/BXWritePC

```

// BXWritePC()
// =====

BXWritePC(bits(32) address)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool() then
            address<1> = '0';
        BranchTo(address, BranchType_UNKNOWN);

```

aarch32/functions/registers/BranchWritePC

```

// BranchWritePC()
// =====

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
        BranchTo(address, BranchType_UNKNOWN);

```

aarch32/functions/registers/D

```

// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    _V[n DIV 2]<base+63:base> = value;
    return;

```

aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// =====

bits(64) Din(integer n)
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

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];
```

aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address);
```

aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    case n of // Select index by mode:    usr fiq irq svc abt und hyp
        when 8    result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9    result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10   result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11   result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12   result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13   result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14   result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise result = n;

    return result;
```

aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

aarch32/functions/registers/PC

```
// PC - non-assignment form
// =====

bits(32) PC
    return R[15]; // This includes the offset from AArch32 state
```


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;
```

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;
```

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];
```

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];
```

aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
    integer svc, integer abt, integer und, integer hyp)
```

```

case mode of
  when M32_User    result = usr; // User mode
  when M32_FIQ    result = fiq; // FIQ mode
  when M32_IRQ    result = irq; // IRQ mode
  when M32_Svc    result = svc; // Supervisor mode
  when M32_Abort  result = abt; // Abort mode
  when M32_Hyp    result = hyp; // Hyp mode
  when M32_Undef  result = und; // Undefined mode
  when M32_System result = usr; // System mode uses User mode registers
  otherwise       Unreachable(); // Monitor mode

return result;

```

aarch32/functions/registers/Rmode

```

// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
assert n >= 0 && n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
  if n == 13 then return SP_mon;
  elseif n == 14 then return LR_mon;
  else return _R[n]<31:0>;
else
  return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
assert n >= 0 && n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
  if n == 13 then SP_mon = value;
  elseif n == 14 then LR_mon = value;
  else _R[n]<31:0> = value;
else
  // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
  // register are unchanged or set to zero. This is also tested for on
  // exception entry, as this applies to all AArch32 registers.
  if !HighestELUsingAArch32() && ConstrainUnpredictableBool() then
    _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
  else
    _R[LookUpRIndex(n, mode)]<31:0> = value;

return;

```

aarch32/functions/registers/S

```

// S[] - non-assignment form
// =====

bits(32) S[integer n]
assert n >= 0 && n <= 31;
base = (n MOD 4) * 32;
return _V[n DIV 4]<base+31:base>;

```

```
// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    _V[n DIV 4]<base+31:base> = value;
    return;
```

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];
```

aarch32/functions/registers/_Dclone

```
array bits(64) _Dclone[0..31];
```

aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    EventRegisterSet();

    // Align PC[1:0] according to the target instruction set state
    if spsr<5> == '1' then // T32
        new_pc = Align(new_pc, 2);
    else // A32
        new_pc = Align(new_pc, 4);

    BranchTo(new_pc, BranchType_UNKNOWN);
```

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;
```

aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and return the register's contents.  
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);
```

aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and return the register's contents.  
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
```

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;
```

aarch32/functions/system/AArch32.SysRegWrite

```
// Write to a 32-bit AArch32 System register.  
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);
```

aarch32/functions/system/AArch32.SysRegWrite64

```
// Write to a 64-bit AArch32 System register.  
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);
```

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,e1) = ELFromM32(mode);  
    if !valid then  
        PSTATE.IL = '1';  
    else  
        PSTATE.M = mode;  
        PSTATE.EL = e1;  
        PSTATE.nRW = '1';  
        PSTATE.SP = if mode IN {M32_User,M32_System} then '0' else '1';  
    return;
```

aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
case mode of
    when M32_User      result = FALSE;
    when M32_FIQ       result = FALSE;
    when M32_IRQ       result = FALSE;
    when M32_Svc       result = FALSE;
    when M32_Monitor   result = !HaveEL(EL3);
    when M32_Abort     result = FALSE;
    when M32_Hyp       result = !HaveEL(EL2);
    when M32_Undef     result = FALSE;
    when M32_System    result = FALSE;
    otherwise          result = TRUE;
return result;
```

aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
    when '000xx', '00100' // R8_usr to R12_usr
        if mode != M32_FIQ then UNPREDICTABLE;
    when '00101' // SP_usr
        if mode == M32_System then UNPREDICTABLE;
    when '00110' // LR_usr
        if mode IN {M32_Hyp, M32_System} then UNPREDICTABLE;
    when '010xx', '0110x', '01110' // R8_fiq to R12_fiq, SP_fiq, LR_fiq
        if mode == M32_FIQ then UNPREDICTABLE;
    when '1000x' // LR_irq, SP_irq
        if mode == M32_IRQ then UNPREDICTABLE;
    when '1001x' // LR_svc, SP_svc
        if mode == M32_Svc then UNPREDICTABLE;
    when '1010x' // LR_abt, SP_abt
        if mode == M32_Abort then UNPREDICTABLE;
    when '1011x' // LR_und, SP_und
        if mode == M32_Undef then UNPREDICTABLE;
    when '1110x' // LR_mon, SP_mon
        if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
    when '11110' // ELR_hyp, only from Monitor or Hyp mode
        if !HaveEL(EL2) || !(mode IN {M32_Monitor, M32_Hyp}) then UNPREDICTABLE;
    when '11111' // SP_hyp, only from Monitor mode
        if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
    otherwise
        UNPREDICTABLE;

return;
```

aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

// Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
```

```

if bytemask<3> == '1' then
    PSTATE.<N,Z,C,V,Q> = value<31:27>;
    // Bits <26:24> are ignored

if bytemask<2> == '1' then
    // Bit <23> is RES0
    if privileged then
        PSTATE.PAN = value<22>;
    // Bits <21:20> are RES0
    PSTATE.GE = value<19:16>;
if bytemask<1> == '1' then
    // Bits <15:10> are RES0
    PSTATE.E = value<9>;
    // PSTATE.E is writable at EL0
    if privileged then
        PSTATE.A = value<8>;

if bytemask<0> == '1' then
    if privileged then
        PSTATE.<I,F> = value<7:6>;
        // Bit <5> is RES0
        // AArch32.WriteMode sets PSTATE.IL to 1 if 'value<4:0>' is not a legal mode value
        AArch32.WriteMode(value<4:0>);

return;

```

aarch32/functions/system/ConditionPassed

```

// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());

```

aarch32/functions/system/CurrentCond

```

bits(4) AArch32.CurrentCond();

```

aarch32/functions/system/InITBlock

```

// InITBlock()
// =====

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;

```

aarch32/functions/system/LastInITBlock

```

// LastInITBlock()
// =====

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');

```

aarch32/functions/system/SPSRWriteByInstr

```

// SPSRWriteByInstr()
// =====

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

new_spsr = SPSR[];

```

```

if bytemask<3> == '1' then
    new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0],J bits

if bytemask<2> == '1' then
    new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags

if bytemask<1> == '1' then
    new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask

if bytemask<0> == '1' then
    new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits

SPSR[] = new_spsr; // UNPREDICTABLE if User or System mode

return;

```

aarch32/functions/system/SPSRAccessValid

```

// SPSRAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRAccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110' // SPSR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000' // SPSR_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '10010' // SPSR_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100' // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110' // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100' // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110' // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;

```

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;

```

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;

```

aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

E1.3.4 aarch32/translation

aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// =====

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits();
        TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.type = MemType_Device;
            memattrs.device = DeviceType_nGnRnE;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.type = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
        when '00010', '00011', '00100'
            // Write-back or Write-through Read allocate, or Non-cacheable
            memattrs.type = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype);
            memattrs.outer = ShortConvertAttrsHints(C:B, acctype);
            memattrs.shareable = (S == '1');
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.type = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints('01', acctype);
            memattrs.outer = ShortConvertAttrsHints('01', acctype);
            memattrs.shareable = (S == '1');
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.type = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype);
            memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype);
            memattrs.shareable = (S == '1');
```



```

otherwise
    // Reserved, handled above
    Unreachable();

// transient bits are not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;

// distinction between inner and outer shareable is not supported in this format
memattrs.outershareable = memattrs.shareable;

return MemAttrDefaults(memattrs);

```

aarch32/translation/attrs/AArch32.InstructionDevice

```

// AArch32.InstructionDevice()
// =====
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress,
                                             bits(40) ipaddress, integer level, bits(4) domain,
                                             AccType acctype, boolean iswrite, boolean secondstage,
                                             boolean s2fs1walk)

c = ConstrainUnpredictable();
assert c IN {Constraint_NONE, Constraint_FAULT};

if c == Constraint_FAULT then
    addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
                                              secondstage, s2fs1walk);
else
    addrdesc.memattrs.type = MemType_Normal;
    addrdesc.memattrs.inner.attrs = MemAttr_NC;
    addrdesc.memattrs.inner.hints = MemHint_No;
    addrdesc.memattrs.outer = addrdesc.memattrs.inner;
    addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

return addrdesc;

```

aarch32/translation/attrs/AArch32.RemappedTEXDecode

```

// AArch32.RemappedTEXDecode()
// =====

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

MemoryAttributes memattrs;

region = UInt(TEX<0>:C:B); // TEX<2:1> are ignored in this mapping scheme
if region == 6 then
    memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
else
    base = 2 * region;
    attrfield = PRRR<base+1:base>;

    if attrfield == '11' then // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    case attrfield of
        when '00' // Device-nGnRnE
            memattrs.type = MemType_Device;
            memattrs.device = DeviceType_nGnRnE;
        when '01' // Device-nGnRE
            memattrs.type = MemType_Device;
            memattrs.device = DeviceType_nGnRE;

```

```

when '10'
    memattrs.type = MemType_Normal;
    memattrs.inner = ShortConvertAttrHints(NMRR<base+1:base>, acctype);
    memattrs.outer = ShortConvertAttrHints(NMRR<base+17:base+16>, acctype);
    s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
    memattrs.shareable = (s_bit == '1');
    memattrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
when '11'
    Unreachable();

// transient bits are not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;

return MemAttrDefaults(memattrs);

```

aarch32/translation/attrs/AArch32.S1AttrDecode

```

// AArch32.S1AttrDecode()
// =====
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

MemoryAttributes memattrs;

if PSTATE.EL == EL2 then
    mair = HMAIR1:HMAIR0;
else
    mair = MAIR1:MAIR0;
index = 8 * UInt(attr);
attrfield = mair<index+7:index>;

if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
    (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
    // Reserved, maps to an allocated value
    (-, attrfield) = ConstrainUnpredictableBits();

if attrfield<7:4> == '0000' then // Device
    memattrs.type = MemType_Device;
    case attrfield<3:0> of
        when '0000' memattrs.device = DeviceType_nGnRnE;
        when '0100' memattrs.device = DeviceType_nGnRE;
        when '1000' memattrs.device = DeviceType_nGRE;
        when '1100' memattrs.device = DeviceType_GRE;
        otherwise Unreachable(); // Reserved, handled above
elseif attrfield<3:0> != '0000' then // Normal
    memattrs.type = MemType_Normal;
    memattrs.outer = LongConvertAttrHints(attrfield<7:4>, acctype);
    memattrs.inner = LongConvertAttrHints(attrfield<3:0>, acctype);
    memattrs.shareable = SH<1> == '1';
    memattrs.outershareable = SH == '10';

else
    Unreachable(); // Reserved, handled above

return MemAttrDefaults(memattrs);

```

aarch32/translation/attrs/AArch32.TranslateAddressS1Off

```

// AArch32.TranslateAddressS1Off()
// =====
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in

```

```
// this pseudocode.

TLBRecord AArch32.TranslateAddressS10ff(bits(32) vaddress, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    default_cacheable = (HasS2Translation() && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) ==
'1'));

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;           // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elseif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR/HSCTLR.I
        if PSTATE.EL == EL2 then
            cacheable = HSCTLR.I == '1';
        else
            cacheable = SCTLR.I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
            result.addrdesc.memattrs.shareable = TRUE;
            result.addrdesc.memattrs.outershareable = TRUE;

        result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

        result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

        result.perms.ap = bits(3) UNKNOWN;
        result.perms.xn = '0';
        result.perms.pxn = '0';

        result.nG = bit UNKNOWN;
        result.contiguous = boolean UNKNOWN;
        result.domain = bits(4) UNKNOWN;
        result.level = integer UNKNOWN;
        result.blocksize = integer UNKNOWN;
        result.addrdesc.paddress.physicaladdress = ZeroExtend(vaddress);
        result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
        result.addrdesc.fault = AArch32.NoFault();

    return result;
```

aarch32/translation/checks/AArch32.CheckDomain

```
// AArch32.CheckDomain()
// =====

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level,
AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;
```

```

if attrfield == '10' then          // Reserved, maps to an allocated value
    // Reserved value maps to an allocated value
    (-, attrfield) = ConstrainUnpredictableBits();

if attrfield == '00' then
    fault = AArch32.DomainFault(domain, level, acctype, iswrite);
else
    fault = AArch32.NoFault();

permissioncheck = (attrfield == '01');

return (permissioncheck, fault);

```

aarch32/translation/checks/AArch32.CheckPermission

```

// AArch32.CheckPermission()
// =====
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                     bits(4) domain, bit NS, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());

if PSTATE.EL != EL2 then
    wxn = SCTL.R.WXN == '1';
    if TTBCR.EAE == '1' || SCTL.R.AFE == '1' || perms.ap<0> == '1' then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';
    else
        priv_r = perms.ap<2:1> != '00';
        priv_w = perms.ap<2:1> == '01';
        user_r = perms.ap<1> == '1';
        user_w = FALSE;
    uwxn = SCTL.R.UWXN == '1';

    if (HavePANExt() && PSTATE.PAN == '1' && user_r && PSTATE.EL != EL0 &&
        !(acctype IN {AccType_DC, AccType_AT, AccType_UNPRIV, AccType_IFETCH})) then
        priv_r = FALSE; priv_w = FALSE;

    user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
    priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
              (priv_w && wxn) || (user_w && uwxn));
    ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

    if ispriv then
        (r, w, xn) = (priv_r, priv_w, priv_xn);
    else
        (r, w, xn) = (user_r, user_w, user_xn);
else
    // Access from EL2
    wxn = HSCTL.R.WXN == '1';
    r = TRUE;
    w = perms.ap<2> == '0';
    xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW } then
        fail = !r || !w;

```

```

        failedread = !r;
    elseif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                         !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();

```

aarch32/translation/checks/AArch32.CheckS2Permission

```

// AArch32.CheckS2Permission()
// =====
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
                                       integer level, AccType acctype, boolean iswrite,
                                       boolean s2fs1walk)

    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    xn = !r || perms.xn == '1';

    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
    elseif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elseif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                         !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();

```

aarch32/translation/debug/AArch32.CheckBreakpoint

```

// AArch32.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

```

```

match = FALSE;
mismatch = FALSE;

for i = 0 to UInt(DBGDIDR.BRPs)
    (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
    match = match || match_i;
    mismatch = mismatch || mismatch_i;

if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Breakpoint;
    Halt(reason);
elseif (match || mismatch) && DBGDSCRExt.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    debugmoe = DebugException_Breakpoint;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();

```

aarch32/translation/debug/AArch32.CheckDebug

```

// AArch32.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch32.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRExt.MDBGGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool();

    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.type == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.type == Fault_None && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;

```

aarch32/translation/debug/AArch32.CheckVectorCatch

```

// AArch32.CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMATCH(vaddress);
    if size == 4 && !match && AArch32.VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool();

```

```

if match && DBGDSCRExt.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    debugmoe = DebugException_VectorCatch;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();

```

aarch32/translation/debug/AArch32.CheckWatchpoint

```

// AArch32.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);

    for i = 0 to UInt(DBGDIDR.WRPs)
        match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elseif match && DBGDSCRExt.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();

```

aarch32/translation/faults/AArch32.AccessFlagFault

```

// AArch32.AccessFlagFault()
// =====

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);

```

aarch32/translation/faults/AArch32.AddressSizeFault

```

// AArch32.AddressSizeFault()
// =====

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);

```

aarch32/translation/faults/AArch32.AlignmentFault

```
// AArch32.AlignmentFault()
// =====

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.AsynchExternalAbort

```
// AArch32.AsynchExternalAbort()
// =====
// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(type, ipaddress, domain, level, acctype, iswrite, extflag,
                                     debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// =====

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Debug, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.DomainFault

```
// AArch32.DomainFault()
// =====

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

    ipaddress = bits(40) UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
```



```
s2fs1walk = FALSE;

return AArch32.CreateFaultRecord(Fault_Domain, ipaddress, domain, level, acctype, iswrite,
                                extflag, debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.NoFault

```
// AArch32.NoFault()
// =====

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_None, ipaddress, domain, level, acctype, iswrite,
                                    extflag, debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.PermissionFault

```
// AArch32.PermissionFault()
// =====

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Permission, ipaddress, domain, level, acctype, iswrite,
                                    extflag, debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/faults/AArch32.TranslationFault

```
// AArch32.TranslationFault()
// =====

FaultRecord AArch32.TranslationFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Translation, ipaddress, domain, level, acctype, iswrite,
                                    extflag, debugmoe, secondstage, s2fs1walk);
```

aarch32/translation/translation/AArch32.FirstStageTranslate

```
// AArch32.FirstStageTranslate()
// =====
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                             boolean wasaligned, integer size)

    if PSTATE.EL == EL2 then
        s1_enabled = HSCTLR.M == '1';
```

```

elseif HaveEL(EL2) && !IsSecure() then
    tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
    dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
    s1_enabled = tge == '0' && dc == '0' && SCTL.R.M == '1';
else
    dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
    s1_enabled = dc == '0' && SCTL.R.M == '1';

ipaddress = bits(40) UNKNOWN;
secondstage = FALSE;
s2fslwalk = FALSE;

if s1_enabled then
    // First stage enabled
    use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
    if use_long_descriptor_format then
        S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                              s2fslwalk, size);
        permissioncheck = TRUE; domaincheck = FALSE;
    else
        S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
        permissioncheck = TRUE; domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE; domaincheck = FALSE;

// Check for unaligned data accesses to Device memory
if (!wasaligned && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
    acctype != AccType_IFETCH) then
    S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
if !IsFault(S1.addrdesc) && domaincheck then
    (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype,
                                                    iswrite);
    S1.addrdesc.fault = abort;

if !IsFault(S1.addrdesc) && permissioncheck then
    S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
                                                S1.domain, S1.addrdesc.paddress.NS,
                                                acctype, iswrite);

// Check for instruction fetches from Device memory not marked as execute-never. If there has
// not been a Permission Fault then the memory is not marked execute-never.
if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
    acctype == AccType_IFETCH) then
    S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                            S1.domain, acctype, iswrite,
                                            secondstage, s2fslwalk);

return S1.addrdesc;

```

aarch32/translation/translation/AArch32.FullTranslate

```

// AArch32.FullTranslate()
// =====
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                       boolean wasaligned, integer size)

// First Stage Translation
S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
if !IsFault(S1) && HasS2Translation() then
    s2fslwalk = FALSE;
    result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fslwalk,
                                          size);
else

```

```
    result = S1;

    return result;
```

aarch32/translation/translation/AArch32.SecondStageTranslate

```
// AArch32.SecondStageTranslate()
// =====
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
                                              AccType acctype, boolean iswrite, boolean wasaligned,
                                              boolean s2fs1walk, integer size)

    assert HasS2Translation();
    assert IsZero(S1.paddress.physicaladdress<47:40>);
    hwupdatewalk = FALSE;
    if !ELUsingAArch32(EL2) then
        return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
                                              wasaligned, s2fs1walk, size, hwupdatewalk);

    s2_enabled = HCR.VM == '1' || HCR.DC == '1';
    secondstage = TRUE;

    if s2_enabled then // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<39:0>;

        S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                              s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype != AccType_IFETCH) then
            S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                         acctype, iswrite, s2fs1walk);
            // Check for instruction fetches from Device memory not marked as execute-never. As there
            // has not been a Permission Fault then the memory is not marked execute-never.
            if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
                acctype == AccType_IFETCH) then
                domain = bits(4) UNKNOWN;
                S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                         domain, acctype, iswrite,
                                                         secondstage, s2fs1walk);

            // Check for protected table walk
            if (s2fs1walk && !IsFault(S2.addrdesc) && HCR.PTW == '1' &&
                S2.addrdesc.memattrs.type == MemType_Device) then
                domain = bits(4) UNKNOWN;
                S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
                                                            iswrite, secondstage, s2fs1walk);

        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

aarch32/translation/translation/AArch32.SecondStageWalk

```
// AArch32.SecondStageWalk()
// =====
// Perform a stage 2 translation on a stage 1 translation page table walk access.
```

```
AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype,
                                          boolean iswrite, integer size)
```

```
    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                       size);
```

aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite,
                                          boolean wasaligned, integer size)

    if !ELUsingAArch32(S1TranslationRegime()) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned,
                                       size);
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

aarch32/translation/walk/AArch32.TranslationTableWalkLD

```
// AArch32.TranslationTableWalkLD()
// =====
// Returns a result of a translation table walk using the Long-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
                                          AccType acctype, boolean iswrite, boolean secondstage,
                                          boolean s2fs1walk, integer size)

    if !secondstage then
        assert ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(40) inputaddr;      // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
    domain = bits(4) UNKNOWN;

    descaddr.memattrs.type = MemType_Normal;

    // Fixed parameters for the page table walk:
    // grainsize = Log2(Size of Table)           - Size of Table is 4KB in AArch32
    // stride = Log2(Address per Level)          - Bits of address consumed at each level
    constant integer grainsize = 12;             // Log2(4KB page size)
    constant integer stride = grainsize - 3;      // Log2(page size / 8 bytes)

    // Derived parameters for the page table walk:
    // inputsize = Log2(Size of Input Address) - Input Address size in bits
```

```
// level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    if PSTATE.EL == EL2 then
        inputsz = 32 - UInt(HTCR.T0SZ);
        basefound = inputsz == 32 || IsZero(inputaddr<31:inputsz>);
        disabled = FALSE;
        baseregister = HTTBR;
        descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGNO, HTCR.IRGNO);
        reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;
    else
        basefound = FALSE;
        disabled = FALSE;
        t0sz = UInt(TTBCR.T0SZ);
        if t0sz == 0 || IsZero(inputaddr<31:(32-t0sz)>) then
            inputsz = 32 - t0sz;
            basefound = TRUE;
            disabled = TTBCR.EPD0 == '1';
            baseregister = TTBR0;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGNO, TTBCR.IRGNO);
        t1sz = UInt(TTBCR.T1SZ);
        if (t1sz == 0 && !basefound) || (t1sz > 0 && IsOnes(inputaddr<31:(32-t1sz)>)) then
            inputsz = 32 - t1sz;
            basefound = TRUE;
            disabled = TTBCR.EPD1 == '1';
            baseregister = TTBR1;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGNO, TTBCR.IRGNO);
            reversedescriptors = SCTLR.EE == '1';
            lookupsecure = IsSecure();
            singlepriv = FALSE;
        // The starting level is the number of strides needed to consume the input address
        level = 4 - RoundUp(Real(inputsz - grainsize) / Real(stride));
    else
        // Second stage translation
        inputaddr = ipaddress;
        inputsz = 32 - UInt(VTCR.T0SZ);
        // VTCR.S must match VTCR.T0SZ[3]
        if VTCR.S != VTCR.T0SZ<3> then
            (-, inputsz) = ConstrainUnpredictableInteger(32-7, 32+8);
        basefound = inputsz == 40 || IsZero(inputaddr<39:inputsz>);
        disabled = FALSE;
        baseregister = VTTBR;
        descaddr.memattrs = WalkAttrDecode(VTCR.IRGNO, VTCR.ORGNO, VTCR.SH0);
        reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

        startlevel = UInt(VTCR.SL0);
        level = 2 - startlevel;
        if level <= 0 then basefound = FALSE;

        // Number of entries in the starting level table =
        // (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
        startsizecheck = inputsz - ((3 - level)*stride + grainsize); // Log2(Num of entries)

        // Check for starting level table with fewer than 2 entries or longer than 16 pages.
        // Lower bound check is: startsizecheck < Log2(2 entries)
        // That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
        // Upper bound check is: startsizecheck > Log2(pagesize/8*16)
        // That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
        if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;
```

```

if !basefound || disabled then
    level = 1; // AArch64 reports this as a level 0 fault
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                    secondstage, s2fs1walk);
    return result;

if !IsZero(baseregister<47:40>) then
    level = 0;
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite,
                                                    secondstage, s2fs1walk);
    return result;

// Bottom bound of the Base address is:
// Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
// (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize); // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.physicaladdress = ZeroExtend(baseaddress OR index);
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if secondstage || !HasS2Translation() then
        descaddr2 = descaddr;
    else
        descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    desc = _Mem[descaddr2, 8, AccType_PTW];
    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
        // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then // Block (01) or Page (11)
        blocktranslate = TRUE;
    else // Table (11)
        if !IsZero(desc<47:40>) then
            result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;

        baseaddress = desc<39:grainsize>:Zeros(grainsize);

        if !secondstage then

```

```

// Unpack the upper and lower table attributes
ns_table = ns_table OR desc<63>;
ap_table<1> = ap_table<1> OR desc<62>; // read-only
xn_table = xn_table OR desc<60>;
// pxn_table and ap_table[0] apply only in EL1&0 translation regimes
if !singlepriv then
    ap_table<0> = ap_table<0> OR desc<61>; // privileged
    pxn_table = pxn_table OR desc<59>;

level = level + 1;
addrselecttop = addrselectbottom - 1;
blocktranslate = FALSE;
until blocktranslate;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
    iswrite, secondstage, s2fslwalk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;

// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
    iswrite, secondstage, s2fslwalk);
    return result;
xn = desc<54>;
pxn = desc<53>;
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
ap = desc<7:6>:'1';
memattr = desc<5:2>; // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN; // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn = xn OR xn_table;
    result.perms.ap<2> = ap<2> OR ap_table<1>; // Force read-only
    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
        result.perms.pxn = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn = '0';
        result.nG = '0';
        result.perms.ap<0> = '1';
        result.addrdesc.memattr = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0> = '1';
    result.perms.xn = xn;
    result.perms.pxn = '0';
    result.nG = '0';
    result.addrdesc.memattr = S2AttrDecode(sh, memattr, acctype);
    result.addrdesc.paddress.NS = '1';

```

```
result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
result.addrdesc.fault = AArch32.NoFault();
result.contiguous = contiguousbit == '1';

return result;
```

aarch32/translation/walk/AArch32.TranslationTableWalkSD

```
// AArch32.TranslationTableWalkSD()
// =====
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    // This is only called when address translation is enabled
    TLBRecord result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;
    bits(40) outputaddress;

    // Variables for Abort functions
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    // Default setting of the domain
    domain = bits(4) UNKNOWN;

    // Determine correct Translation Table Base Register to use.
    bits(64) ttbr;
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(vaddress<31:(32-n)>) then
        ttbr = TTBR0;
        disabled = (TTBCR.PD0 == '1');
    else
        ttbr = TTBR1;
        disabled = (TTBCR.PD1 == '1');
        n = 0; // TTBR1 translation always works like N=0 TTBR0 translation

    // Check this Translation Table Base Register is not disabled.
    if disabled then
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Obtain descriptor from initial lookup.
    l1descaddr.paddress.physicaladdress = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
    l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
    IRGN = ttbr<0>:ttbr<6>; // TTBR.IRGN
    RGN = ttbr<4:3>; // TTBR.RGN
    SH = ttbr<1>:ttbr<5>; // TTBR.S:TTBR.NOS
    l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN);

    if !HaveEL(EL2) || IsSecure() then
        // if only 1 stage of translation
        l1descaddr2 = l1descaddr;
    else
        l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
        // Check for a fault on the stage 2 walk
```



```

    if IsFault(l1descaddr2) then
        result.addrdesc.fault = l1descaddr2.fault;
        return result;

// Update virtual address for abort functions
l1descaddr2.vaddress = ZeroExtend(vaddress);

l1desc = _Mem[l1descaddr2, 4, AccType_PTW];
if SCTL.R.EE == '1' then l1desc = BigEndianReverse(l1desc);

// Process descriptor from initial lookup.
case l1desc<1:0> of
    when '00' // Fault, Reserved
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
            iswrite, secondstage, s2fslwalk);
        return result;
    when '01' // Large page or Small page
        domain = l1desc<8:5>;
        level = 2;
        pxn = l1desc<2>;
        NS = l1desc<3>;

        // Obtain descriptor from level 2 lookup.
        l2descaddr.paddress.physicaladdress = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
        l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
        l2descaddr.memattrs = l1descaddr.memattrs;

        if !HaveEL(EL2) || IsSecure() then
            // if only 1 stage of translation
            l2descaddr2 = l2descaddr;
        else
            l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
            // Check for a fault on the stage 2 walk
            if IsFault(l2descaddr2) then
                result.addrdesc.fault = l2descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        l2descaddr2.vaddress = ZeroExtend(vaddress);

        l2desc = _Mem[l2descaddr2, 4, AccType_PTW];
        if SCTL.R.EE == '1' then l2desc = BigEndianReverse(l2desc);

        // Process descriptor from level 2 lookup.
        if l2desc<1:0> == '00' then
            result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                iswrite, secondstage, s2fslwalk);
            return result;

        nG = l2desc<11>;
        S = l2desc<10>;
        ap = l2desc<9,5:4>;

        if SCTL.R.AFE == '1' && l2desc<4> == '0' then
            // Hardware access to the Access Flag is not supported in ARMv8
            result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                iswrite, secondstage, s2fslwalk);
            return result;

        if l2desc<1> == '0' then // Large page
            xn = l2desc<15>;
            tex = l2desc<14:12>;
            c = l2desc<3>;
            b = l2desc<2>;
            blocksize = 64;
            outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);

```

```

else // Small page
    tex = l2desc<8:6>;
    c = l2desc<3>;
    b = l2desc<2>;
    xn = l2desc<0>;
    blocksize = 4;
    outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);

when '1x' // Section or Supersection
    NS = l1desc<19>;
    nG = l1desc<17>;
    S = l1desc<16>;
    ap = l1desc<15,11:10>;
    tex = l1desc<14:12>;
    xn = l1desc<4>;
    c = l1desc<3>;
    b = l1desc<2>;
    pxn = l1desc<0>;
    level = 1;

    if SCTL.R.AFE == '1' && l1desc<10> == '0' then
        // Hardware management of the Access Flag is not supported in ARMv8
        result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
            iswrite, secondstage, s2fslwalk);

        return result;

    if l1desc<18> == '0' then // Section
        domain = l1desc<8:5>;
        blocksize = 1024;
        outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
    else // Supersection
        domain = '0000';
        blocksize = 16384;
        outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

// Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
if SCTL.R.TRE == '0' then
    if RemapRegsHaveResetValues() then
        result.addrdesc.memattr = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
    else
        result.addrdesc.memattr = MemoryAttributes IMPLEMENTATION_DEFINED;
else
    result.addrdesc.memattr = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

// Set the rest of the TLBRecord, try to add it to the TLB, and return it.
result.perms.ap = ap;
result.perms.xn = xn;
result.perms.pxn = pxn;
result.nG = nG;
result.domain = domain;
result.level = level;
result.blocksize = blocksize;
result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
result.addrdesc.fault = AArch32.NoFault();

return result;

```

aarch32/translation/walk/RemapRegsHaveResetValues

```
boolean RemapRegsHaveResetValues();
```

E1.4 Common library pseudocode

E1.4.1 shared/debug

shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// =====

ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RXO = '0';           // Clear RX overrun flag
    if Halted() then           // in Debug state
        EDSCR.ITO = '0';       // Clear ITR overrun flag
    EDSCR.ERR = '0';           // Clear cumulative error flag
    return;
```

shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =====

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =====
// Returns the debug exception target Exception level

bits(2) DebugTargetFrom(boolean secure)

    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elseif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
```

```

if ELUsingAArch32(EL1) then
    return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();

```

shared/debug/FindWatchpoint/FindWatchpoint

```

// FindWatchpoint()
// =====

integer FindWatchpoint()
    address = FAR[];
    base = Align(address, ZVAGranuleSize());
    limit = base + ZVAGranuleSize();
    repeat
        for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
            if WatchpointByteMatch(i, address) then // Candidate found
                return i;
            address = address + 1;
            if address == limit then address = base; // Wrap round, as this must be a DC ZVA
    while address != FAR[];
    return -1; // No candidate found (should not happen)

```

shared/debug/authentication/AllowExternalDebugAccess

```

// AllowExternalDebugAccess()
// =====
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalInvasiveDebugEnabled() then
        if ExternalSecureInvasiveDebugEnabled() then
            return TRUE;
        elseif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;

```

shared/debug/authentication/AllowExternalPMUAccess

```

// AllowExternalPMUAccess()
// =====
// Returns the status of EDPRSR.EPMAD.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalNoninvasiveDebugEnabled() then
        if ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        elseif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;

```

shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// =====

boolean ExternalInvasiveDebugEnabled()
    // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN
    // signal.
    return DBGEN == HIGH;
```

shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =====

boolean ExternalNoninvasiveDebugEnabled()
    // Return TRUE if Trace and Sample-based profiling are allowed
    return (ExternalNoninvasiveDebugEnabled() &&
        (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
        (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUIDEN == '1')));
```

shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =====

boolean ExternalNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
    // OR NIDEN) signal.
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// =====

boolean ExternalSecureInvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state of the
    // (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====

boolean ExternalSecureNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the state of the
    // (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```

shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.  
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.  
CTI_SignalEvent(CrossTriggerIn id);
```

shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,  
                             CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,  
                             CrossTriggerOut_TraceExtIn0,    CrossTriggerOut_TraceExtIn1,  
                             CrossTriggerOut_TraceExtIn2,    CrossTriggerOut_TraceExtIn3};  
  
enumeration CrossTriggerIn {CrossTriggerIn_CrossHalt,      CrossTriggerIn_PMUOverflow,  
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,  
                             CrossTriggerIn_TraceExtOut0,   CrossTriggerIn_TraceExtOut1,  
                             CrossTriggerIn_TraceExtOut2,   CrossTriggerIn_TraceExtOut3};
```

shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()  
// =====  
  
CheckForDCCInterrupts()  
    commrx = (EDSCR.RXfull == '1');  
    commtx = (EDSCR.TXfull == '0');  
  
    // COMMRX and COMMTX support is optional and not recommended for new designs.  
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);  
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);  
  
    // The value to be driven onto the common COMMIRQ signal.  
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||  
              (commtx && MDCCINT_EL1.TX == '1'));  
    SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then HIGH else LOW);  
  
    return;
```

shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)  
// =====  
// Called on writes to debug register 0x08C.  
  
DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value  
  
    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits  
        IMPLEMENTATION_DEFINED "signal slave-generated error";  
        return;  
  
    if EDSCR.ERR == '1' then return; // Error flag set: ignore write  
  
    // The Software lock is OPTIONAL.  
    if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write  
  
    if EDSCR.RXfull == '1' || (Halted\(\) && EDSCR.MA == '1' && EDSCR.ITE == '0') then  
        EDSCR.RX0 = '1'; EDSCR.ERR = '1'; // Overrun condition: ignore write  
        return;  
  
    EDSCR.RXfull = '1';  
    DTRRX = value;
```

```

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)

    if !UsingAArch32() then
        ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
        X[1] = bits(64) UNKNOWN;
    else
        ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
        ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
        R[1] = bits(32) UNKNOWN;

    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(32) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';

    EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// =====

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
return DTRRX;

```

shared/debug/dccanditr/DBGDTRTX_EL0

```

// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN;

underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
value = if underrun then bits(32) UNKNOWN else DTRTX;

if EDSCR.ERR == '1' then return value; // Error flag set: no side-effects

// The Software Lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then // Software lock locked: no side-effects
    return value;

if underrun then
    EDSCR.TXU = '1'; EDSCR.ERR = '1'; // Underrun condition: block side-effects
    return value; // Return UNKNOWN

EDSCR.TXfull = '0';

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)

    if !UsingAArch32() then
        ExecuteA64(0xB8404401<31:0>); // A64 "LDR W1,[X0],#4"
    else
        ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"

    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then

```

```

    EDSCR.TXfull = bit UNKNOWN;
    DBGDTRTX_EL0 = bits(32) UNKNOWN;
else
    if !UsingAArch32() then
        ExecuteA64(0xD5130501<31:0>);          // A64 "MSR DBGDTRTX_EL0,X1"
    else
        ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
        // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
        assert EDSCR.TXfull == '1';

    if !UsingAArch32() then
        X[1] = bits(64) UNKNOWN;
    else
        R[1] = bits(32) UNKNOWN;

    EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// =====

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;

```

shared/debug/dccanditr/DBGDTR_EL0

```

// DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
// For MSR DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
// For MSR DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
assert N IN {32,64};
if EDSCR.TXfull == '1' then
    value = bits(N) UNKNOWN;
// On a 64-bit write, implement a half-duplex channel
if N == 64 then DTRRX = value<63:32>;
DTRTX = value<31:0>; // 32-bit or 64-bit write
EDSCR.TXfull = '1';
return;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
// For MRS <Rt>,DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
// For MRS <Xt>,DBGDTR_EL0 N=64, X[t]=result
assert N IN {32,64};
bits(N) result;
if EDSCR.RXfull == '0' then
    result = bits(N) UNKNOWN;
else
    // On a 64-bit read, implement a half-duplex channel
    // NOTE: the word order is reversed on reads with regards to writes
    if N == 64 then result<63:32> = DTRTX;
    result<31:0> = DTRRX;
EDSCR.RXfull = '0';
return result;

```


shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// =====
// Called on writes to debug register 0x088.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

    if !Halted() then return;                      // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1'; EDSCR.ERR = '1';          // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrComp1" 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 "InstrComp1".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0> /*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elseif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then UndefinedFault();
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UndefinedFault();
            elseif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elseif IsSecure() then UndefinedFault();
            else handle_el = EL2;

        when EL3
```

```

        if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
        handle_el = EL3;
        from_secure = IsSecure();
        if ELUsingAArch32(handle_el) then
            if PSTATE.M == M32_Monitor then SCR.NS = '0';
            assert UsingAArch32(); // Cannot move from AArch64 to AArch32
            case handle_el of
                when EL1
                    AArch32.WriteMode(M32_Svc);
                    if HavePANExt() && SCTL.R.SPAN == '0' then
                        PSTATE.PAN = '1';
                when EL2 AArch32.WriteMode(M32_Hyp);
                when EL3
                    AArch32.WriteMode(M32_Monitor);
                    if HavePANExt() then
                        if !from_secure then
                            PSTATE.PAN = '0';
                        elsif 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;
        else // Targeting AArch64
            if UsingAArch32() then AArch64.MaybeZeroRegisterUppers();
            ELR[] = bits(64) UNKNOWN; SPSR[] = bits(32) UNKNOWN; ESR[] = bits(32) UNKNOWN;
            PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;

        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(32) UNKNOWN;
        UpdateEDSCRFIELDS(); // Update EDSCR processor state flags.

    return;

```

shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()
    SynchronizeContext();
    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(32) UNKNOWN;

    UpdateEDSCRFIELDS(); // Update EDSCR processor state flags.

    return;

```

shared/debug/halting/DebugHalt

```

constant bits(6) DebugHalt_Breakpoint = '000111';
constant bits(6) DebugHalt_EDBGCRQ   = '010011';
constant bits(6) DebugHalt_StepNormal = '011011';

```

```
constant bits(6) DebugHalt_Step_Exclusive = '011111';
constant bits(6) DebugHalt_OSUnlockCatch = '100011';
constant bits(6) DebugHalt_ResetCatch    = '100111';
constant bits(6) DebugHalt_Watchpoint    = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess = '110011';
constant bits(6) DebugHalt_ExceptionCatch = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// =====

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the processor is restarting, debuggers must use EDPRSR.SDR
    // to detect that the processor has restarted.
    EDSCR.STATUS = '000001'; // Signal restarting
    EDESr<2:0> = '000';       // Clear any pending Halting debug events

    new_pc = DLR_EL0;
    spsr = DSPSR;

    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0.

    if UsingAArch32() then
        if ConstrainUnpredictableBool() then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN); // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool() then
            new_pc<63:32> = Zeros();
            BranchTo(new_pc, BranchType_DBGEXIT); // A type of branch that is never predicted

        (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
        UpdateEDSCRFields(); // Stop signalling processor state.
        DisableITRAndResumeInstructionPrefetch();

    return;
```

shared/debug/halting/Halt

```
// Halt()
// =====

Halt(bits(6) reason)
```

```

CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt

DLR_EL0 = ThisInstrAddr();
DPSR_EL0 = GetPSRFromPSTATE();
DPSR_EL0.SS = PSTATE.SS; // Always save PSTATE.SS

EDSCR.ITE = '1'; EDSCR.ITO = '0';
if IsSecure() then
    EDSCR.SDD = '0'; // If entered in Secure state, allow debug
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';

// PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
// UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
// exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
// unchanged. PSTATE.IL is set to 0.
if UsingAArch32() then
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
else
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

StopInstructionPrefetchAndEnableITR();
EDSCR.STATUS = reason; // Signal entered Debug state
UpdateEDSCRFields(); // Update EDSCR processor state flags.

return;

```

shared/debug/halting/HaltOnBreakpointOrWatchpoint

```

// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';

```

shared/debug/halting/Halted

```

// Halted()
// =====

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'}); // Halted

```

shared/debug/halting/HaltingAllowed

```

// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elseif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();

```

shared/debug/halting/Restarting

```
// Restarting()
// =====

boolean Restarting()
    return EDSCR.STATUS == '000001'; // Restarting
```

shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR processor state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';
        EDSCR.RW<1> = (if ELUsingAArch32(EL1) then '0' else '1');
        if PSTATE.EL != EL0 then
            EDSCR.RW<0> = EDSCR.RW<1>;
        else
            EDSCR.RW<0> = (if UsingAArch32() then '0' else '1');
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0') then
            EDSCR.RW<2> = EDSCR.RW<1>;
        else
            EDSCR.RW<2> = (if ELUsingAArch32(EL2) then '0' else '1');
        if !HaveEL(EL3) then
            EDSCR.RW<3> = EDSCR.RW<2>;
        else
            EDSCR.RW<3> = (if ELUsingAArch32(EL3) then '0' else '1');

    return;
```

shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch()
    // Called after taking an exception, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() && EDECCR<UInt>(PSTATE.EL) + base == '1' then
        Halt(DebugHalt_ExceptionCatch);
```

shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
```

```

if HaltingStep_DidNotStep() then
    Halt(DebugHalt_Step_NoSyndrome);
elseif HaltingStep_SteppedEX() then
    Halt(DebugHalt_Step_Exclusive);
else
    Halt(DebugHalt_Step_Normal);

```

shared/debug/haltingevents/CheckOSUnlockCatch

```

// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

```

```

CheckOSUnlockCatch()
    if EDECR.OSUC == '1' && !Halted() then EDESR.OSUC = '1';

```

shared/debug/haltingevents/CheckPendingOSUnlockCatch

```

// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

```

```

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);

```

shared/debug/haltingevents/CheckPendingResetCatch

```

// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

```

```

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);

```

shared/debug/haltingevents/CheckResetCatch

```

// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);

```

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);

```

shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

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();
```

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();
```

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" == 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" = 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
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

shared/debug/interrupts/InterruptID

```
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
                        InterruptID_COMMRX, InterruptID_COMMTX};
```

shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// =====

CreatePCSample()
// In a simple sequential execution of the program, CreatePCSample is executed each time the PE
// executes an instruction that can be sampled. An implementation is not constrained such that
// reads of EDPCSRlo return the current values of PC, etc.

pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
pc_sample.pc = ThisInstrAddr();
pc_sample.el = PSTATE.EL;
pc_sample.rw = if UsingAArch32() then '0' else '1';
pc_sample.ns = if IsSecure() then '0' else '1';
pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
if HaveEL(EL2) && !IsSecure() then
    if ELUsingAArch32(EL2) then
        pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
    elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        pc_sample.vmid = VTTBR_EL2.VMID;
if HaveVirtHostExt() && !IsSecure() && !ELUsingAArch32(EL2) then
    pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
else
    pc_sample.contextidr_el2 = bits(32) UNKNOWN;
return;
```

shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =====

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        if HaveVirtHostExt() && EDSCR.SC2 == '1' then
            EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            EDPCSRhi.EL = pc_sample.el;
            EDPCSRhi.NS = pc_sample.ns;
        else
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDVIDSR = pc_sample.contextidr_el2;
            else
                EDVIDSR.VMID = (if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1, EL0}
                    then pc_sample.vmid else Zeros(16));
                EDVIDSR.NS = pc_sample.ns;
                EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                // The conditions for setting HV are not specified if PCSRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
```



```
EDPCSRhi = bits(32) UNKNOWN;
EDCIDSR = bits(32) UNKNOWN;
EDVIDSR = bits(32) UNKNOWN;
```

```
return sample;
```

shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    bits(16) vmid
)
```

```
PCSample pc_sample;
```

shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then AArch64.SoftwareStepException();
```

shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elseif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

    if IllegalExceptionReturn(spsr) then
        dest = PSTATE.EL;
    else
        (valid, dest) = ELFromSPSR(spsr); assert valid;

    secure = IsSecureBelowEL3() || dest == EL3;

    if ELUsingAArch32(dest) then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
```

```

    ELd = DebugTargetFrom(secure);
    if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;

    return SS_bit;

```

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) && MDCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;

```

shared/debug/softwarestep/SoftwareStep_DidNotStep

```

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean SoftwareStep_DidNotStep();

```

shared/debug/softwarestep/SoftwareStep_SteppedEX

```

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean SoftwareStep_SteppedEX();

```

E1.4.2 shared/exceptions

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() 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;

```

shared/exceptions/exceptions/Exception

```

enumeration Exception {Exception_Uncategorized, // Uncategorized or unknown reason
    Exception_WFxTrap, // Trapped WFI or WFE instruction
    Exception_CP15RRTTrap, // Trapped AArch32 MCR or MRC access to CP15
    Exception_CP15RRRTTrap, // Trapped AArch32 MCRR or MRRC access to CP15
    Exception_CP14RTTrap, // Trapped AArch32 MCR or MRC access to CP14
    Exception_CP14DTTrap, // Trapped AArch32 LDC or STC access to CP14
    Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap, // Trapped access to SIMD or FP ID register
    // Trapped BXJ instruction not supported in ARMv8
    Exception_CP14RRRTTrap, // Trapped MRRC access to CP14 from AArch32
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_MonitorCall, // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCAlignment, // Misaligned PC
    Exception_DataAbort, // Data Abort
    Exception_SPAAlignment, // Misaligned SP
    Exception_FPtrappedException, // IEEE trapped FP exception
    Exception_SError, // SError interrupt or Asynchronous Abort
    Exception_Breakpoint, // (Hardware) Breakpoint
    Exception_SoftwareStep, // Software Step
    Exception_Watchpoint, // Watchpoint
    Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
    Exception_VectorCatch, // AArch32 Vector Catch
    Exception_IRQ, // IRQ interrupt
    Exception_FIQ; // FIQ interrupt
}

```

shared/exceptions/exceptions/ExceptionRecord

```

type ExceptionRecord is (Exception type, // Exception class
    bits(25) syndrome, // Syndrome record
    bits(64) vaddress, // Virtual fault address
    boolean ipavalid, // Physical fault address is valid
    bits(48) ipaddress) // Physical fault address for second stage faults

```

shared/exceptions/exceptions/ExceptionSyndrome

```

// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

```

```
ExceptionRecord ExceptionSyndrome(Exception type)
```

```
ExceptionRecord r;
```

```

r.type = type;

// Initialize all other fields
r.syndrome = Zeros();
r.vaddress = Zeros();
r.ipavalid = FALSE;
r.ipaddress = Zeros();

return r;

```

shared/exceptions/traps/ReservedValue

```

// ReservedValue()
// =====

ReservedValue()

if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
    AArch32.TakeUndefInstrException();
else
    AArch64.UndefinedFault();

```

shared/exceptions/traps/UnallocatedEncoding

```

// UnallocatedEncoding()
// =====

UnallocatedEncoding()

// If the unallocated encoding is an AArch32 CP10 or CP11 instruction, FPEXC.DEX must be written
// to zero. This is omitted from this code.
if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
    AArch32.TakeUndefInstrException();
else
    AArch64.UndefinedFault();

```

E1.4.3 shared/functions

shared/functions/aborts/EncodeLDFSC

```

// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault type, integer level)

bits(6) result;
case type of
    when Fault_AddressSize      result = '0000':level<1:0>; assert level IN {0,1,2,3};
    when Fault_AccessFlag       result = '0010':level<1:0>; assert level IN {1,2,3};
    when Fault_Permission        result = '0011':level<1:0>; assert level IN {1,2,3};
    when Fault_Translation       result = '0001':level<1:0>; assert level IN {0,1,2,3};
    when Fault_SyncExternal      result = '010000';
    when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
    when Fault_SyncParity        result = '011000';
    when Fault_SyncParityOnWalk  result = '0111':level<1:0>; assert level IN {0,1,2,3};
    when Fault_AsyncParity       result = '011001';
    when Fault_AsyncExternal     result = '010001';
    when Fault_Alignment         result = '100001';
    when Fault_Debug             result = '100010';
    when Fault_TLBConflict       result = '110000';
    when Fault_Lockdown          result = '110100'; // IMPLEMENTATION DEFINED
    when Fault_Exclusive         result = '110101'; // IMPLEMENTATION DEFINED

```

```

        otherwise
            Unreachable();

    return result;

```

shared/functions/aborts/FaultSyndrome

```

// FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode or an Exception Level using AArch64.

bits(25) FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(25) iss = Zeros();
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.type, fault.level);

    return iss;

```

shared/functions/aborts/IPAValid

```

// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault_None;

    if fault.s2fs1walk then
        return fault.type IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                               Fault_AddressSize};
    elseif fault.secondstage then
        return fault.type IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;

```

shared/functions/aborts/IsAsyncAbort

```

// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.type);

```

shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.type == Fault_Debug;
```

shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_AsyncExternal, Fault_AsyncParity,
                    Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk});

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.type);
```

shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;
```

shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;

    return fault.secondstage;
```

shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```

shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

shared/functions/common/ASR_C

```
// ASR_C()
// =====

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

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;
```

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>;
```

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;
```

shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// =====

integer CountLeadingZeroBits(bits(N) x)
    return N - 1 - HighestSetBit(x);
```

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;
```

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);
```

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;
```


shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

shared/functions/common/LSL_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

shared/functions/common/LSR

```
// LSR()
// =====
```

```
bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

shared/functions/common/LSR_C

```
// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

shared/functions/common/LeastSetBit

```
// LeastSetBit()
// =====

integer LeastSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

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;
```

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;
```

shared/functions/common/NOT

```
bits(N) NOT(bits(N) x);
```

shared/functions/common/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// =====

bits(N) Ones()
    return Ones(N);
```

shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

shared/functions/common/ROR_C

```
// ROR_C()
// =====

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

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);
```

shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

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);
```

shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

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 + 2i;
    if x<N-1> == '1' then result = result - 2N;
    return result;
```

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);
```

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 + 2i;
    return result;
```

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);
```

shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0', N);
```

```
// Zeros()
// =====

bits(N) Zeros()
    return Zeros(N);
```

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;
```

shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// =====

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR poly:Zeros(i-32);
    return data<31:0>;
```

shared/functions/crypto/AESInvMixColumns

```
bits(128) AESInvMixColumns(bits (128) op);
```

shared/functions/crypto/AESInvShiftRows

```
bits(128) AESInvShiftRows(bits(128) op);
```

shared/functions/crypto/AESInvSubBytes

```
bits(128) AESInvSubBytes(bits(128) op);
```

shared/functions/crypto/AESMixColumns

```
bits(128) AESMixColumns(bits (128) op);
```

shared/functions/crypto/AESShiftRows

```
bits(128) AESShiftRows(bits(128) op);
```

shared/functions/crypto/AESSubBytes

```
bits(128) AESSubBytes(bits(128) op);
```

shared/functions/crypto/HaveCryptoExt

```
boolean HaveCryptoExt();
```

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);
```

shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

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);
```

shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// =====

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// =====

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

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));
```

shared/functions/crypto/SHAparity

```
// SHAparity()
// =====

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusive Monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at address.
// It is IMPLEMENTATION DEFINED whether the global Exclusive Monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress address, integer processorid, integer size);
```

shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusive Monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// =====

// Clear the local Exclusive Monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

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();
```

shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusive Monitor for processorid includes all of
// the physical address region of size bytes starting at address.
boolean IsExclusiveGlobal(FullAddress address, integer processorid, integer size);
```

shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusive Monitor for processorid includes all of
// the physical address region of size bytes starting at address.
boolean IsExclusiveLocal(FullAddress address, integer processorid, integer size);
```

shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusive Monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusive Monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// =====

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveHPDExt

```
// HaveHPDExt()
// =====

boolean HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =====

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HavePANExt

```
// HavePANExt()
// =====

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// =====

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitMechanismExt()
    return HasArchVersion(ARMv8p1);
```


shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =====

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// =====

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRTYPE fpcr, FPRounding rounding)
    assert N IN {32,64};
    assert M IN {32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

bits(N) FPAbs(bits(N) op)
    assert N IN {32,64};
    return '0' : op<N-2:0>;
```

shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE_Infinity); inf2 = (type2 == FPTYPE_Infinity);
        zero1 = (type1 == FPTYPE_Zero); zero2 = (type2 == FPTYPE_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        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 // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr, rounding);
return result;

```

shared/functions/float/fpcompare/FPCompare

```

// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRTType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTYPE_SNaN || type1==FPTYPE_QNaN || type2==FPTYPE_SNaN || type2==FPTYPE_QNaN then
        result = '0011';
        if type1==FPTYPE_SNaN || type2==FPTYPE_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elseif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';
    return result;

```

shared/functions/float/fpcompareeq/FPCompareEQ

```

// FPCompareEQ()
// =====

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTYPE_SNaN || type1==FPTYPE_QNaN || type2==FPTYPE_SNaN || type2==FPTYPE_QNaN then
        result = FALSE;
        if type1==FPTYPE_SNaN || type2==FPTYPE_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;

```

shared/functions/float/fpcomparege/FPCompareGE

```

// FPCompareGE()
// =====

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTYPE_SNaN || type1==FPTYPE_QNaN || type2==FPTYPE_SNaN || type2==FPTYPE_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else

```

```

    // All non-NaN cases can be evaluated on the values produced by FPUntpack()
    result = (value1 >= value2);
    return result;

```

shared/functions/float/fpcomparegt/FPCmpareGT

```

// FPCmpareGT()
// =====

boolean FPCmpareGT(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);
    if type1==FPTType_SNaN || type1==FPTType_QNaN || type2==FPTType_SNaN || type2==FPTType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 > value2);
    return result;

```

shared/functions/float/fpconvert/FPConvert

```

// FPConvert()
// =====

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.

bits(M) FPConvert(bits(N) op, FPCRTType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (type,sign,value) = FPUntpack(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if type == FPTType_SNaN || type == FPTType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elseif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if type == FPTType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp, fpcr);
    elseif type == FPTType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elseif type == FPTType_Zero then
        result = FPZero(sign);
    else
        result = FPRound(value, fpcr, rounding);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRTType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));

```

shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// =====

// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

shared/functions/float/fpcrtype/FPCRTType

```
type FPCRTType;
```

shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecoderM()
// =====

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF; // P
        when '11' return FPRounding_NEGINF; // M
```

shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// =====

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF; // P
        when '10' return FPRounding_NEGINF; // M
        when '11' return FPRounding_ZERO; // Z
```

shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// =====

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = '0';
    exp = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =====

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPinfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

shared/functions/float/fpexc/FPExc

```
enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
    FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// =====

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

shared/functions/float/fpmax/FPMMax

```
// FPMMax()
// =====

bits(N) FPMMax(bits(N) op1, bits(N) op2, FPCRTType fpcr)
```

```

assert N IN {32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    if value1 > value2 then
        (type,sign,value) = (type1,sign1,value1);
    else
        (type,sign,value) = (type2,sign2,value2);
    if type == FPType_Infinity then
        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        sign = sign1 AND sign2; // Use most positive sign
        result = FPZero(sign);
    else
        result = FPRound(value, fpcr);
return result;

```

shared/functions/float/fpmaxnormal/FPMaxNormal

```

// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - E - 1;
exp = Ones(E-1):'0';
frac = Ones(F);
return sign : exp : frac;

```

shared/functions/float/fpmaxnum/FPMaxNum

```

// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
assert N IN {32,64};

(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

// treat a single quiet-NaN as -Infinity
if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('1');
elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('1');

return FPMAX(op1, op2, fpcr);

```

shared/functions/float/fpmin/FPMin

```

// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
assert N IN {32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    if value1 < value2 then
        (type,sign,value) = (type1,sign1,value1);
    else
        (type,sign,value) = (type2,sign2,value2);
    if type == FPType_Infinity then

```

```

        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        sign = sign1 OR sign2; // Use most negative sign
        result = FPZero(sign);
    else
        result = FPRound(value, fpcr);
    return result;

```

shared/functions/float/fpminnum/FPMinNum

```

// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
    assert N IN {32,64};

    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');

    return FPMIn(op1, op2, fpcr);

```

shared/functions/float/fpmul/FPMul

```

// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;

```

shared/functions/float/fpmuladd/FPMulAdd

```

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

```

```

inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
(done,result) = FPPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN();
    FPPProcessException(FPExc_InvalidOp, fpcr);

if !done then
    infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

    // Determine sign and type product will have if it does not cause an Invalid
    // Operation.
    signP = sign1 EOR sign2;
    infP = inf1 || inf2;
    zeroP = zero1 || zero2;

    // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
    // additions of opposite-signed infinities.
    if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
        result = FPDefaultNaN();
        FPPProcessException(FPExc_InvalidOp, fpcr);

    // Other cases involving infinities produce an infinity of the same sign.
    elseif (infA && signA == '0') || (infP && signP == '0') then
        result = FPInfinity('0');
    elseif (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.
    elseif zeroA && zeroP && signA == signP then
        result = FPZero(signA);

    // Otherwise calculate numerical result and round it.
    else
        result_value = valueA + (value1 * value2);
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
            result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(result_sign);
        else
            result = FPRound(result_value, fpcr);

return result;

```

shared/functions/float/fpmulx/FPMuIX

```

// FPMuIX()
// =====

bits(N) FPMuIX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elseif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);

```



```

else
    result = FPRound(value1*value2, fpcr);
return result;

```

shared/functions/float/fpneg/FPNeg

```

// FPNeg()
// =====

bits(N) FPNeg(bits(N) op)
    assert N IN {32,64};
    return NOT(op<N-1>) : op<N-2:0>;

```

shared/functions/float/fponepointfive/FPOnePointFive

```

// FPOnePointFive()
// =====

bits(N) FPOnePointFive(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;

```

shared/functions/float/fpprocessexception/FPProcessException

```

// FPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRTYPE fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp      cumul = 0;
        when FPExc_DivideByZero    cumul = 1;
        when FPExc_Overflow        cumul = 2;
        when FPExc_Underflow       cumul = 3;
        when FPExc_Inexact         cumul = 4;
        when FPExc_InputDenorm     cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elseif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;

```

shared/functions/float/fpprocesNaN/FPProcessNaN

```

// FPProcessNaN()
// =====

bits(N) FPProcessNaN(FPTYPE type, bits(N) op, FPCRTYPE fpcr)
    assert N IN {32,64};
    assert type IN {FPTYPE_QNaN, FPTYPE_SNaN};

```

```

topfrac = if N == 32 then 22 else 51;
result = op;
if type == FPType_SNaN then
    result<topfrac> = '1';
    FPProcessException(FPExc_InvalidOp, fpcr);
if fpcr.DN == '1' then // DefaultNaN requested
    result = FPDefaultNaN();
return result;

```

shared/functions/float/fpprocessnans/FPProcessNaNs

```

// FPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                bits(N) op1, bits(N) op2,
                                FPCRTYPE fpcr)

assert N IN {32,64};
if type1 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elseif type1 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
else
    done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);

```

shared/functions/float/fpprocessnans3/FPProcessNaNs3

```

// FPProcessNaNs3()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRTYPE fpcr)

assert N IN {32,64};
if type1 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elseif type3 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
elseif type1 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elseif type3 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type3, op3, fpcr);

```

```
else
    done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```

shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =====

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {32, 64};
    (type,sign,value) = FPUntpack(operand, fpcr);
    if type == FPUntpack_SNaN || type == FPUntpack_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elseif type == FPUntpack_Infinity then
        result = FPZero(sign);
    elseif type == FPUntpack_Zero then
        result = FPinfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elseif (N == 32 && Abs(value) < 2.0^-128)
        || (N == 64 && Abs(value) < 2.0^-1024) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPinfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elseif fpcr.FZ == '1'
        && ((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);
        FPProcessException(FPExc_Underflow, fpcr);
    else
        // Scale to a double-precision value in the range 0.5 <= x < 1.0, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction extended with zeros.

        if N == 32 then
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        else // N == 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';
        scaled = '0' : '0111111110' : fraction<51:44> : Zeros(44);

        if N == 32 then
            result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
        else // N == 64
            result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // Call C function to get reciprocal estimate of scaled value.
        // Input is rounded down to a multiple of 1/512.
```

```

estimate = recip_estimate(scaled);

// Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
// Convert to scaled single-precision result with copied sign bit and high-order
// fraction bits, and exponent calculated above.

fraction = estimate<51:0>;
if result_exp == 0 then
    fraction = '1' : fraction<51:1>;
elseif result_exp == -1 then
    fraction = '01' : fraction<51:2>;
    result_exp = 0;
if N == 32 then
    result = sign : result_exp<N-25:0> : fraction<51:29>;
else // N == 64
    result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;

```

shared/functions/float/fprecp/FPRecpX

```

// FPRecpX()
// =====

bits(N) FPRecpX(bits(N) op, FPCRTType fpcr)
    assert N IN {32,64};
    bits(N) result;
    integer esize = if N == 32 then 8 else 11;
    bits(esize) exp;
    bits(esize) max_exp;
    bits(N-esize-1) frac = Zeros();

    if N == 32 then
        exp = op<23+esize-1:23>;
    else
        exp = op<52+esize-1:52>;
    max_exp = Ones(esize) - 1;

    (type,sign,value) = FPUncpack(op, fpcr);
    if type == FPTType_SNaN || type == FPTType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;

```

shared/functions/float/fpround/FPRound

```

// FPRound()
// =====

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRound(real op, FPCRTType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14; E = 5; F = 10;

```

```

elseif N == 32 then
    minimum_exp = -126; E = 8; F = 23;
else // N == 64
    minimum_exp = -1022; E = 11; F = 52;

// Split value into sign, unrounded mantissa and exponent.
if op < 0.0 then
    sign = '1'; mantissa = -op;
else
    sign = '0'; mantissa = op;
exponent = 0;
while mantissa < 1.0 do
    mantissa = mantissa * 2.0; exponent = exponent - 1;
while mantissa >= 2.0 do
    mantissa = mantissa / 2.0; exponent = exponent + 1;

// Deal with flush-to-zero.
if fpcr.FZ == '1' && N != 16 && exponent < minimum_exp then
    // Flush-to-zero never generates a trapped exception
    if UsingAArch32() then
        FPSCR.UFC = '1';
    else
        FPSR.UFC = '1';
    return FPZero(sign);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max(exponent - minimum_exp + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped.
if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
        overflow_to_inf = TRUE;
    when FPRounding_POSINF
        round_up = (error != 0.0 && sign == '0');
        overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
        round_up = (error != 0.0 && sign == '1');
        overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD
        round_up = FALSE;
        overflow_to_inf = FALSE;

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then // Rounded up to next exponent
        biased_exp = biased_exp + 1; int_mant = int_mant DIV 2;

// Handle rounding to odd aka Von Neumann rounding
if error != 0.0 && rounding == FPRounding_ODD then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then

```

```

        result = if overflow_to_inf then FPinfinity(sign) else FPMMaxNormal(sign);
        FPPProcessException(FPExc_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
    else // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0; // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

    // Deal with Inexact exception.
    if error != 0.0 then
        FPPProcessException(FPExc_Inexact, fpcr);

    return result;

// FPRound()
// =====

bits(N) FPRound(real op, FPCRTYPE fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));

```

shared/functions/float/fprounding/FPRounding

```

enumeration FPRounding {FPRounding_TIEEVEN, FPRounding_POSINF,
                        FPRounding_NEGINF, FPRounding_ZERO,
                        FPRounding_TIEAWAY, FPRounding_ODD};

```

shared/functions/float/fproundingmode/FPRoundingMode

```

// FPRoundingMode()
// =====

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRTYPE fpcr)
    return FPDecodeRounding(fpcr.RMode);

```

shared/functions/float/fproundint/FPRoundInt

```

// FPRoundInt()
// =====

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRTYPE fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    if type == FPTYPE_SNaN || type == FPTYPE_QNaN then
        result = FPPProcessNaN(type, op, fpcr);
    elseif type == FPTYPE_Infinity then
        result = FPinfinity(sign);
    elseif type == FPTYPE_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

```

```
// Determine whether supplied rounding mode requires an increment
case rounding of
  when FPRounding_TIEEVEN
    round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
  when FPRounding_POSINF
    round_up = (error != 0.0);
  when FPRounding_NEGINF
    round_up = FALSE;
  when FPRounding_ZERO
    round_up = (error != 0.0 && int_result < 0);
  when FPRounding_TIEAWAY
    round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

if round_up then int_result = int_result + 1;

// Convert integer value into an equivalent real value
real_result = Real(int_result);

// Re-encode as a floating-point value, result is always exact
if real_result == 0.0 then
  result = FPZero(sign);
else
  result = FPRound(real_result, fpcr, FPRounding_ZERO);

// Generate inexact exceptions
if error != 0.0 && exact then
  FPProcessException(FPExc_Inexact, fpcr);

return result;
```

shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =====

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRTYPE fpcr)
  assert N IN {32, 64};
  (type,sign,value) = FPUnpack(operand, fpcr);
  if type == FPTYPE_SNaN || type == FPTYPE_QNaN then
    result = FPProcessNaN(type, operand, fpcr);
  elseif type == FPTYPE_Zero then
    result = FPInfinity(sign);
    FPProcessException(FPExc_DivideByZero, fpcr);
  elseif sign == '1' then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elseif type == FPTYPE_Infinity then
    result = FPZero('0');
  else
    // Scale to a double-precision value in the range 0.25 <= x < 1.0, 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.

    if N == 32 then
      fraction = operand<22:0> : Zeros(29);
      exp = UInt(operand<30:23>);
    else // N == 64
      fraction = operand<51:0>;
      exp = UInt(operand<62:52>);

    if exp == 0 then
      while fraction<51> == 0 do
        fraction = fraction<50:0> : '0';
        exp = exp - 1;
      fraction = fraction<50:0> : '0';
```

```

if exp<0> == '0' then
    scaled = '0' : '0111111110' : fraction<51:44> : Zeros(44);
else
    scaled = '0' : '0111111101' : fraction<51:44> : Zeros(44);

if N == 32 then
    result_exp = (380 - exp) DIV 2;
else // N == 64
    result_exp = (3068 - exp) DIV 2;

// Call C function to get reciprocal estimate of scaled value.
estimate = recip_sqrt_estimate(scaled);

// Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
// Convert to scaled single-precision result with copied sign bit and high-order
// fraction bits, and exponent calculated above.

if N == 32 then
    result = '0' : result_exp<N-25:0> : estimate<51:29>;
else // N == 64
    result = '0' : result_exp<N-54:0> : estimate<51:0>;
return result;

```

shared/functions/float/fpsqrt/FPSqrt

```

// FPSqrt()
// =====

bits(N) FPSqrt(bits(N) op, FPCRTType fpcr)
assert N IN {32,64};
(type,sign,value) = FPUunpack(op, fpcr);
if type == FPType_SNaN || type == FPType_QNaN then
    result = FPProcessNaN(type, op, fpcr);
elseif type == FPType_Zero then
    result = FPZero(sign);
elseif type == FPType_Infinity && sign == '0' then
    result = FPInfinity(sign);
elseif sign == '1' then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    result = FPRound(Sqrt(value), fpcr);
return result;

```

shared/functions/float/fpsub/FPSub

```

// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
assert N IN {32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUunpack(op1, fpcr);
(type2,sign2,value2) = FPUunpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    if inf1 && inf2 && sign1 == sign2 then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
        result = FPInfinity('0');

```



```

elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPInfinity('1');
elseif zero1 && zero2 && sign1 == NOT(sign2) then
    result = FPZero(sign1);
else
    result_value = value1 - value2;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr, rounding);
return result;

```

shared/functions/float/fpthree/FPThree

```

// FPThree()
// =====

bits(N) FPThree(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;

```

shared/functions/float/fptofixed/FPToFixed

```

// FPToFixed()
// =====

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRTYPE fpcr, FPRounding rounding)
    assert N IN {32,64};
    assert M IN {32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUntpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if type == FPTYPE_SNaN || type == FPTYPE_QNaN then
        FPPROCESSException(FPEXC_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

```

```
// Generate saturated result and exceptions
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPExc_InvalidOp, fpcr);
elseif error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);

return result;
```

shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

bits(N) FPTwo(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                        FPType_QNaN, FPType_SNaN};
```

shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRTYPE fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign = fpval<15>;
        exp16 = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero
            if IsZero(frac16) then
                type = FPType_Zero; value = 0.0;
            else
                type = FPType_Nonzero; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elseif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                type = FPType_Infinity; value = 2.0^1000000;
            else
                type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elseif N == 32 then
```

```

sign = fpval<31>;
exp32 = fpval<30:23>;
frac32 = fpval<22:0>;
if IsZero(exp32) then
    // Produce zero if value is zero or flush-to-zero is selected.
    if IsZero(frac32) || fpcr.FZ == '1' then
        type = FPType_Zero; value = 0.0;
        if !IsZero(frac32) then // Denormalized input flushed to zero
            FPProcessException(FPExc_InputDenorm, fpcr);
    else
        type = FPType_Nonzero; value = 2.0-126 * (Real(UInt(frac32)) * 2.0-23);
elseif IsOnes(exp32) then
    if IsZero(frac32) then
        type = FPType_Infinity; value = 2.01000000;
    else
        type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
        value = 0.0;
else
    type = FPType_Nonzero;
    value = 2.0(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0-23);

else // N == 64

    sign = fpval<63>;
    exp64 = fpval<62:52>;
    frac64 = fpval<51:0>;
    if IsZero(exp64) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac64) || fpcr.FZ == '1' then
            type = FPType_Zero; value = 0.0;
            if !IsZero(frac64) then // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
        else
            type = FPType_Nonzero; value = 2.0-1022 * (Real(UInt(frac64)) * 2.0-52);
    elseif IsOnes(exp64) then
        if IsZero(frac64) then
            type = FPType_Infinity; value = 2.01000000;
        else
            type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        type = FPType_Nonzero;
        value = 2.0(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0-52);

    if sign == '1' then value = -value;
    return (type, sign, value);

```

shared/functions/float/fpzero/FPZero

```

// FPZero()
// =====

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;

```

shared/functions/float/vfpexpandimm/VFPEExpandImm

```

// VFPEExpandImm()
// =====

```

```
bits(N) VFPEExpandImm(bits(8) imm8)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    return sign : exp : frac;
```

shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,           // Normal loads and stores
    AccType_STREAM, AccType_VECSTREAM, // Streaming loads and stores
    AccType_ATOMIC, AccType_ATOMICRW, // Atomic loads and stores
    AccType_ORDERED, AccType_ORDEREDRW, // Load-Acquire and Store-Release
    AccType_LIMITEDORDERED,           // Load-LOAcquire and Store-LORelease
    AccType_UNPRIV,                   // Load and store unprivileged
    AccType_IFETCH,                   // Instruction fetch
    AccType_PTW,                      // Page table walk
    // Other operations
    AccType_DC,                       // Data cache maintenance
    AccType_IC,                       // Instruction cache maintenance
    AccType_AT};                      // Address translation
```

shared/functions/memory/AddrTop

```
// AddrTop()
// =====

integer AddrTop(bits(64) address, bits(2) el)
    // Return the MSB number of a virtual address in the current stage 1 translation
    // regime. If EL1 is using AArch64 then addresses from EL0 using AArch32
    // are zero-extended to 64 bits.
    if UsingAArch32() && !(el == EL0 && !ELUsingAArch32(EL1)) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case el of
            when EL0, EL1
                tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            when EL2
                tbi = TCR_EL2.TBI;
            when EL3
                tbi = TCR_EL3.TBI;
        return (if tbi == '1' then 55 else 63);
```

shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord    fault,      // fault.type indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress     address,
    bits(64)        vaddress
)
```

shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';    // No allocate
constant bits(2) MemHint_WA = '01';    // Write-allocate, Read-no-allocate
constant bits(2) MemHint_RA = '10';    // Read-allocate, Write-no-allocate
constant bits(2) MemHint_RWA = '11';   // Read-allocate and Write-allocate
```

shared/functions/memory/BigEndian

```
// BigEndian()
// =====

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elseif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// =====

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

shared/functions/memory/BranchAddr

```
// BranchAddr()
// =====
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, el);
    if msbit == 63 then
        return vaddress;
    elseif el IN {EL0, EL1} && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';    // Non-cacheable
constant bits(2) MemAttr_WT = '10';    // Write-through
constant bits(2) MemAttr_WB = '11';    // Write-back
```

shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (  
    boolean AF,           // AF needs to be set  
    boolean AP,           // AP[2] / HAP[2] will be modified  
    AddressDescriptor descaddr // Descriptor to be updated  
)
```

shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

shared/functions/memory/Fault

```
enumeration Fault {Fault_None,  
    Fault_AccessFlag,  
    Fault_Alignment,  
    Fault_Background,  
    Fault_Domain,  
    Fault_Permission,  
    Fault_Translation,  
    Fault_AddressSize,  
    Fault_SyncExternal,  
    Fault_SyncExternalOnWalk,  
    Fault_SyncParity,  
    Fault_SyncParityOnWalk,  
    Fault_AsyncParity,  
    Fault_AsyncExternal,  
    Fault_Debug,  
    Fault_TLBConflict,  
    Fault_Lockdown,  
    Fault_Exclusive,  
    Fault_ICacheMaint};
```

shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault type,           // Fault Status  
    AccType acctype,       // Type of access that faulted  
    bits(48) ipaddress,    // Intermediate physical address  
    boolean s2fs1walk,     // Is on a Stage 1 page table walk  
    boolean write,         // TRUE for a write, FALSE for a read  
    integer level,         // For translation, access flag and permission faults  
    bit extflag,           // IMPLEMENTATION DEFINED syndrome for external aborts  
    boolean secondstage,   // Is a Stage 2 abort  
    bits(4) domain,       // Domain number, AArch32 only  
    bits(4) debugmoe)     // Debug method of entry, from AArch32 only
```

shared/functions/memory/FullAddress

```
type FullAddress is (  
    bits(48) physicaladdress,  
    bit NS           // '0' = Secure, '1' = Non-secure  
)
```

shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory accesses
// when they do occur, such as pre-loading the the specified address into one or more caches as
// indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint stream.
// Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a synchronous
// abort due to alignment or translation faults and the like. Its only effect on software visible
// state should be on caches and TLBs associated with address, which must be accessible by reads,
// writes or execution as defined in the translation regime of the current Exception level.
// It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                          MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs, // The possible encodings for each attributes field are as below
    bits(2) hints, // The possible encodings for the hints are below
    boolean transient
)
```

shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType type,
    DeviceType device, // For Device memory types
    MemAttrHints inner, // Inner hints and attributes
    MemAttrHints outer, // Outer hints and attributes
    boolean shareable,
    boolean outershareable
)
```

shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap, // Access permission bits
    bit xn, // Execute-never bit
    bit pxn // Privileged execute-never bit
)
```

shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions perms,
    bit nG,          // '0' = Global, '1' = not Global
    bits(4) domain,  // AArch32 only
    boolean contiguous, // Contiguous bit from page table
    integer level,    // In AArch32 Short-descriptort format, indicates Section/Page
    integer blocksize, // Describes size of memory translated in KBytes
    DescriptorUpdate descupdate, // [ARMv8.1A] Context for hardware update of translation table
    descriptor
    AddressDescriptor addrdesc
)
```

shared/functions/memory/_Mem

```
// These two _Mem[] accessors are the hardware operations which perform
// single-copy atomic, aligned, little-endian memory accesses of size
// bytes from/to the underlying physical memory array of bytes.
//
// The functions address the array using desc.PADDRESS which supplies:
//
// * A 48-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of
//   the array.
//
// The acctype parameter describes the access type: normal, exclusive,
// ordered, streaming, etc.
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccType acctype];

_Mem[AddressDescriptor desc, integer size, AccType acctype] = bits(8*size) value;
```

shared/functions/registers/BranchTo

```
// BranchTo()
// =====

// Set program counter to a new address, which may include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = BranchAddr(target<63:0>, PSTATE.EL);
    return;
```

shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// =====

// Set program counter to a new address, which does not include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        return;
```



```
    assert N == 64 && !UsingAArch32();
    _PC = target<63:0>;
    return;
```

shared/functions/registers/BranchType

```
enumeration BranchType {BranchType_CALL, BranchType_ERET, BranchType_DBGEXIT,
    BranchType_RET, BranchType_JMP, BranchType_EXCEPTION,
    BranchType_UNKNOWN};
```

shared/functions/registers/Hint_Branch

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

shared/functions/registers/NextInstrAddr

```
// Return address of the next instruction.
bits(N) NextInstrAddr();
```

shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

shared/functions/registers/_PC

```
bits(64) _PC;
```

shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

shared/functions/registers/_V

```
array bits(128) _V[0..31];
```

shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// =====

bits(32) SPSR[]
    bits(32) result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ    result = SPSR_fiq;
            when M32_IRQ    result = SPSR_irq;
            when M32_Svc    result = SPSR_svc;
            when M32_Monitor result = SPSR_mon;
```

```

        when M32_Abort    result = SPSR_abt;
        when M32_Hyp      result = SPSR_hyp;
        when M32_Undef    result = SPSR_und;
        otherwise        Unreachable();
    else
        case PSTATE.EL of
            when EL1      result = SPSR_EL1;
            when EL2      result = SPSR_EL2;
            when EL3      result = SPSR_EL3;
            otherwise     Unreachable();

    return result;

// SPSR[] - assignment form
// =====

SPSR[] = bits(32) value
    if UsingArch32() then
        case PSTATE.M of
            when M32_FIQ    SPSR_fiq = value;
            when M32_IRQ    SPSR_irq = value;
            when M32_Svc    SPSR_svc = value;
            when M32_Monitor SPSR_mon = value;
            when M32_Abort  SPSR_abt = value;
            when M32_Hyp    SPSR_hyp = value;
            when M32_Undef  SPSR_und = value;
            otherwise      Unreachable();
        else
            case PSTATE.EL of
                when EL1    SPSR_EL1 = value;
                when EL2    SPSR_EL2 = value;
                when EL3    SPSR_EL3 = value;
                otherwise   Unreachable();

    return;

```

shared/functions/system/ArchVersion

```

enumeration ArchVersion {
    ARMv8p0,
    ARMv8p1,
};

```

shared/functions/system/ClearEventRegister

```

ClearEventRegister();

```

shared/functions/system/ConditionHolds

```

// ConditionHolds()
// =====

// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');           // EQ or NE
        when '001' result = (PSTATE.C == '1');           // CS or CC
        when '010' result = (PSTATE.N == '1');           // MI or PL
        when '011' result = (PSTATE.V == '1');           // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);       // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' result = TRUE;                          // AL

```

```
// Condition flag values in the set '111x' indicate always true
// Otherwise, invert condition if necessary.
if cond<0> == '1' && cond != '1111' then
    result = !result;

return result;
```

shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =====

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

shared/functions/system/CurrentPL

```
// CurrentPL()
// =====

PrivilegeLevel CurrentPL()
    return PLOFEL(PSTATE.EL);
```

shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

// Convert an AArch32 mode encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'mode<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'mode'.

(boolean,bits(2)) ELFromM32(bits(5) mode)
    bits(2) el;
    boolean valid = TRUE;
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = !HaveEL(EL3) || SCR_GEN[].NS == '1';
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            el = if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1;
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;
    if valid then valid = HaveAArch32EL(el);
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then // AArch64 state
        e1 = spsr<3:2>;
        if HighestELUsingAArch32() then // No AArch64 support
            valid = FALSE;
        elseif !HaveEL(e1) then // Exception level not implemented
            valid = FALSE;
        elseif spsr<1> == '1' then // M[1] must be 0
            valid = FALSE;
        elseif e1 == EL0 && spsr<0> == '1' then // for EL0, M[0] must be 0
            valid = FALSE;
        elseif e1 == EL2 && HaveEL(EL3) && SCR_EL3.NS == '0' then // EL2 only valid in Non-secure state
            valid = FALSE;
        else
            valid = TRUE;
    elseif !HaveAnyAArch32() then // AArch32 not supported
        valid = FALSE;
    else // AArch32 state
        (valid, e1) = ELFromM32(spsr<4:0>);
        if !valid then e1 = bits(2) UNKNOWN;
        return (valid,e1);
```

shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====

boolean ELStateUsingAArch32(bits(2) e1, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'e1 IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(e1, secure);
    assert known;
    return aarch32;
```

shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// =====

(boolean,boolean) ELStateUsingAArch32K(bits(2) e1, boolean secure)
    // Returns (known, aarch32):
    // 'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    // using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    // 'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    boolean aarch32;
    known = TRUE;
    if !HaveAArch32EL(e1) then // All levels are using AArch64
        aarch32 = FALSE;
    elseif HighestELUsingAArch32() then // All levels are using AArch32
        aarch32 = TRUE;
    else
        aarch32_below_e13 = HaveEL(EL3) && SCR_EL3.RW == '0';
        aarch32_at_e11 = (aarch32_below_e13 || (HaveEL(EL2) && !secure && HCR_EL2.RW == '0' &&
            !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt())));
        if e1 == EL0 && !aarch32_at_e11 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);

```

shared/functions/system/ELUsingAArch32

```

// ELUsingAArch32()
// =====

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());

```

shared/functions/system/ELUsingAArch32K

```

// ELUsingAArch32K()
// =====

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());

```

shared/functions/system/EndOfInstruction

```

// Terminate processing of the current instruction.
EndOfInstruction();

```

shared/functions/system/EventRegisterSet

```

// Set the local event register in this PE.
EventRegisterSet();

```

shared/functions/system/EventRegistered

```

boolean EventRegistered();

```

shared/functions/system/GetPSRFromPSTATE

```

// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<21>    = PSTATE.SS;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        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
        spsr<9:6>   = PSTATE.<D,A,I,F>;

```

```

    spsr<4>      = PSTATE.nRW;
    spsr<3:2>    = PSTATE.EL;
    spsr<0>      = PSTATE.SP;
    return spsr;

```

shared/functions/system/HasArchVersion

```

// HasArchVersion()
// =====

// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;

```

shared/functions/system/HaveAArch32EL

```

// HaveAArch32EL()
// =====

boolean HaveAArch32EL(bits(2) e1)
    // Return TRUE if Exception level 'e1' supports AArch32
    if !HaveEL(e1) then
        return FALSE;
    elseif !HaveAnyAArch32() then
        return FALSE; // No exception level can use AArch32
    elseif HighestELUsingAArch32() then
        return TRUE; // All exception levels must use AArch32
    elseif e1 == EL0 then
        return TRUE; // EL0 must support using AArch32
    return boolean IMPLEMENTATION_DEFINED;

```

shared/functions/system/HaveAnyAArch32

```

// HaveAnyAArch32()
// =====
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;

```

shared/functions/system/HaveEL

```

// HaveEL()
// =====
// Return TRUE if Exception level 'e1' is supported

boolean HaveEL(bits(2) e1)
    if e1 IN {EL1, EL0} then
        return TRUE; // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;

```

shared/functions/system/HighestEL

```

// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elseif HaveEL(EL2) then

```

```
    return EL2;
else
    return EL1;
```

shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =====
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;    // e.g. CFG32SIGNAL == HIGH
```

shared/functions/system/Hint_Debug

```
Hint_Debug(bits(4) option);
```

shared/functions/system/Hint_Yield

```
Hint_Yield();
```

shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)

    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for return to EL1 in Non-secure state when HCR_EL2.TGE is set
    if target == EL1 && !IsSecureBelowEL3() && HCR_EL2.TGE == '1' then return TRUE;

    return FALSE;
```

shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

shared/functions/system/InterruptPending

```
boolean InterruptPending();
```

shared/functions/system/IsInHost

```
// IsInHost()
// =====
// Returns TRUE if HaveVirtHostExt() is TRUE and executing within a Host OS or an EL0 application
// of a Host OS using AArch64 with HCR_EL2.E2H set to 1, and FALSE otherwise.

boolean IsInHost()
    return (!IsSecure() && HaveVirtHostExt() && !ELUsingAArch32(EL2) &&
        HCR_EL2.E2H == '1' && (PSTATE.EL == EL2 || HCR_EL2.TGE == '1'));
```

shared/functions/system/IsSecure

```
// IsSecure()
// =====

boolean IsSecure()
    // Return TRUE if current Exception level is in Secure state.
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elseif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// =====

// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_GEN[].NS == '0';
    elseif HaveEL(EL2) then
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure;
        return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/Mode_Bits

```
constant bits(5) M32_User      = '10000';
constant bits(5) M32_FIQ      = '10001';
constant bits(5) M32_IRQ      = '10010';
```



```
constant bits(5) M32_Svc      = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

shared/functions/system/PLOfEL

```
// PLOfEL()
// =====

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3 return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2 return PL2;
        when EL1 return PL1;
        when EL0 return PL0;
```

shared/functions/system/PSTATE

```
ProcState PSTATE;
```

shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

shared/functions/system/ProcState

```
type ProcState is (
    bits(1) N,      // Negative condition flag
    bits(1) Z,      // Zero condition flag
    bits(1) C,      // Carry condition flag
    bits(1) V,      // oVerflow condition flag
    bits(1) D,      // Debug mask bit [AArch64 only]
    bits(1) A,      // Asynchronous abort mask bit
    bits(1) I,      // IRQ mask bit
    bits(1) F,      // FIQ mask bit
    bits(1) PAN,    // Privileged Access Never Bit [v8.1-A]
    bits(1) SS,     // Software step bit
    bits(1) IL,     // Illegal Execution state bit
    bits(2) EL,     // Exception Level
    bits(1) nRW,    // not Register Width: 0=64, 1=32
    bits(1) SP,     // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits(1) Q,      // Cumulative saturation flag [AArch32 only]
    bits(4) GE,     // Greater than or Equal flags [AArch32 only]
    bits(8) IT,     // If-then bits, RES0 in CPSR [AArch32 only]
    bits(1) J,      // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
    bits(1) T,      // T32 bit, RES0 in CPSR [AArch32 only]
    bits(1) E,      // Endianness bit [AArch32 only]
    bits(5) M,      // Mode field [AArch32 only]
)
```

shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
    it = spsr<15:10,26:25>;

// When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
// to zero or copied from the SPSR.
```

```

if PSTATE.IL == '1' then
    if ConstrainUnpredictableBool() 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;

```

shared/functions/system/SCRType

```
type SCRType;
```

shared/functions/system/SCR_GEN

```

// SCR_GEN[]
// =====

SCRType SCR_GEN[]
// AArch32 secure & AArch64 EL3 registers are not architecturally mapped
assert HaveEL(EL3);
bits(32) r;
if HighestELUsingAArch32() then
    r = SCR;
else
    r = SCR_EL3;
return r;

```

shared/functions/system/SendEvent

```

// Signal an event to all PEs.
SendEvent();

```

shared/functions/system/SetPSTATEFromPSR

```

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)

SynchronizeContext();

PSTATE.SS = DebugExceptionReturnSS(spsr);

if IllegalExceptionReturn(spsr) then
    PSTATE.IL = '1';
else
    // State that is reinstated only on a legal exception return
    PSTATE.IL = spsr<20>;
    if spsr<4> == '1' then // AArch32 state
        AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
    else // AArch64 state
        PSTATE.nRW = '0';
        PSTATE.EL = spsr<3:2>;
        PSTATE.SP = spsr<0>;

// If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether

```

```
// the T bit is set to zero or copied from SPSR.
if PSTATE.IL == '1' && PSTATE.nRW == '1' then
    if ConstrainUnpredictableBool() then spsr<5> = '0';

// State that is reinstated regardless of illegal exception return
PSTATE.<N,Z,C,V> = spsr<31:28>;
if PSTATE.nRW == '1' then // AArch32 state
    PSTATE.Q = spsr<27>;
    PSTATE.IT = RestoredITBits(spsr);
    PSTATE.GE = spsr<19:16>;
    PSTATE.E = spsr<9>;
    PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
    PSTATE.T = spsr<5>; // PSTATE.J is RES0
else // AArch64 state
    PSTATE.<D,A,I,F> = spsr<9:6>; // No PSTATE.<Q,IT,GE,E,T> in AArch64 state

if HavePANExt() then PSTATE.<PAN> = spsr<22>;

return;
```

shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

shared/functions/system/UsingAArch32

```
// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;
```

shared/functions/system/WaitForEvent

```
WaitForEvent();
```

shared/functions/system/WaitForInterrupt

```
WaitForInterrupt();
```

shared/functions/unpredictable/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.)
Constraint ConstrainUnpredictable();
```

shared/functions/unpredictable/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.
(Constraint,bits(width)) ConstrainUnpredictableBits();
```

shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// =====

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

boolean ConstrainUnpredictableBool()

    c = ConstrainUnpredictable();
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

shared/functions/unpredictable/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.
(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high);
```

shared/functions/unpredictable/Constraint

```
enumeration Constraint    { // General:
    Constraint_NONE, Constraint_UNKNOWN,
    Constraint_UNDEF, Constraint_NOP,
    Constraint_TRUE, Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND, Constraint_COND, Constraint_ADDITIONAL_DECODE,
    // Load-store:
    Constraint_WBSUPPRESS, Constraint_FAULT,
    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

shared/functions/vector/AdvSIMDEExpandImm

```
// AdvSIMDEExpandImm()
// =====

bits(64) AdvSIMDEExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
```

```

        imm64 = Replicate(Zeros(8):imm8, 4);
    when '101'
        imm64 = Replicate(imm8:Zeros(8), 4);
    when '110'
        if cmode<0> == '0' then
            imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
        else
            imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
    when '111'
        if cmode<0> == '0' && op == '0' then
            imm64 = Replicate(imm8, 8);
        if cmode<0> == '0' && op == '1' then
            imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
            imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
            imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
            imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
            imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
        if cmode<0> == '1' && op == '0' then
            imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5>:0:Zeros(19);
            imm64 = Replicate(imm32, 2);
        if cmode<0> == '1' && op == '1' then
            if UsingArch32() then ReservedEncoding();
            imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5>:0:Zeros(48);

    return imm64;

```

shared/functions/vector/PolynomialMult

```

// PolynomialMult()
// =====

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;

```

shared/functions/vector/SatQ

```

// SatQ()
// =====

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);

```

shared/functions/vector/SignedSatQ

```

// SignedSatQ()
// =====

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1; saturated = TRUE;
    elseif i < -(2^(N-1)) then
        result = -(2^(N-1)); saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);

```

shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

bits(32) UnsignedRSqrtEstimate(bits(32) operand)

    if operand<31:30> == '00' then // Operands <= 0xFFFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
        // Generate double-precision value = operand * 2Λ(-32). This has zero sign bit, with:
        //     exponent = 1022 or 1021 = double-precision representation of 2Λ(-1) or 2Λ(-2)
        //     fraction taken from operand, excluding its most significant one or two bits.
        if operand<31> == '1' then
            dp_operand = '0 0111111110' : operand<30:0> : Zeros(21);
        else // operand<31:30> == '01'
            dp_operand = '0 0111111101' : operand<29:0> : Zeros(22);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2Λ31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

return result;
```

shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

bits(32) UnsignedRecipEstimate(bits(32) operand)

    if operand<31> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
        // Generate double-precision value = operand * 2Λ(-32). This has zero sign bit, with:
        //     exponent = 1022 = double-precision representation of 2Λ(-1)
        //     fraction taken from operand, excluding its most significant bit.
        dp_operand = '0 0111111110' : operand<30:0> : Zeros(21);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2Λ31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

return result;
```

shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// =====

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2ΛN - 1 then
        result = 2ΛN - 1; saturated = TRUE;
    elsif i < 0 then
        result = 0; saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
```

E1.4.4 shared/translation**shared/translation/attrs/CombineS1S2AttrHints**

```
// CombineS1S2AttrHints()
// =====

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;

    if s2desc.attrs == '01' || s1desc.attrs == '01' then
        result.attrs = bits(2) UNKNOWN; // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
        result.attrs = MemAttr_NC; // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
        result.attrs = MemAttr_WT; // Write-through
    else
        result.attrs = MemAttr_WB; // Write-back

    result.hints = s1desc.hints;
    result.transient = s1desc.transient;

    return result;
```

shared/translation/attrs/CombineS1S2Desc

```
// CombineS1S2Desc()
// =====
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        elsif s2desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                s2desc.memattrs.device);
    else // Both Normal
        result.memattrs.type = MemType_Normal;
        result.memattrs.device = DeviceType UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
        result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
        result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
            s2desc.memattrs.outershareable);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// =====
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elseif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elseif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =====
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);

    MemAttrHints result;

    if S1CacheDisabled(acctype) then // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elseif attrfield<3:0> == '0100' then // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elseif attrfield<3:2> == '01' then // Write-back transient
            result.attrs = attrfield<1:0>;
            result.hints = MemAttr_WB;
            result.transient = TRUE;
        else // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```

shared/translation/attrs/MemAttrDefaults

```
// MemAttrDefaults()
// =====
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)

    if memattrs.type == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
```



```

        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;

    return memattrs;

```

shared/translation/attrs/S1CacheDisabled

```

// S1CacheDisabled()
// =====

boolean S1CacheDisabled(AccType acctype)
    if ELUsingAArch32(S1TranslationRegime()) then
        if PSTATE.EL == EL2 then
            enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
        else
            enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[.I] else SCTLR[.C];

    return enable == '0';

```

shared/translation/attrs/S2AttrDecode

```

// S2AttrDecode()
// =====
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;

    if attr<3:2> == '00' then // Device
        memattrs.type = MemType_Device;
        case attr<1:0> of
            when '00' memattrs.device = DeviceType_nGnRnE;
            when '01' memattrs.device = DeviceType_nGnRE;
            when '10' memattrs.device = DeviceType_nGRE;
            when '11' memattrs.device = DeviceType_GRE;

    elseif attr<1:0> != '00' then // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        memattrs = MemoryAttributes UNKNOWN; // Reserved

    return MemAttrDefaults(memattrs);

```

shared/translation/attrs/S2CacheDisabled

```

// S2CacheDisabled()
// =====

boolean S2CacheDisabled(AccType acctype)
    if ELUsingAArch32(EL2) then
        disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
    else

```

```

        disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;

    return disable == '1';

```

shared/translation/attrs/S2ConvertAttrsHints

```

// S2ConvertAttrsHints()
// =====
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    if S2CacheDisabled(acctype) then // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case attr of
            when '01' // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '10' // Write-through
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RWA;
            when '11' // Write-back
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;

    result.transient = FALSE;

    return result;

```

shared/translation/attrs/ShortConvertAttrsHints

```

// ShortConvertAttrsHints()
// =====
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype)

    MemAttrHints result;

    if S1CacheDisabled(acctype) then // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00' // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01' // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10' // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11' // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

```

```
result.transient = FALSE;

return result;
```

shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// =====

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN)

    MemoryAttributes memattrs;

    AccType acctype = AccType_NORMAL;

    memattrs.type = MemType_Normal;
    memattrs.inner = ShortConvertAttrHints(IRGN, acctype);
    memattrs.outer = ShortConvertAttrHints(ORGN, acctype);
    memattrs.shareable = SH<1> == '1';
    memattrs.outershareable = SH == '10';

    return MemAttrDefaults(memattrs);
```

shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// =====
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (HaveEL(EL2) && !IsSecure()) && !IsInHost() && PSTATE.EL IN {EL0,EL1});
```

shared/translation/translation/Have16bitVMID

```
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.
boolean Have16bitVMID();
```

shared/translation/translation/PAMax

```
// PAMax()
// =====
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()

    case ID_AA64MMFR0_EL1.PARange of
        when '0000' pa_size = 32;
        when '0001' pa_size = 36;
        when '0010' pa_size = 40;
        when '0011' pa_size = 42;
        when '0100' pa_size = 44;
        when '0101' pa_size = 48;
        otherwise Unreachable();

    return pa_size;
```

shared/translation/translation/S1TranslationRegime

```
// 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()
    if PSTATE.EL != EL0 then
        return PSTATE.EL;
    elsif IsSecure() && HaveEL(EL3) && ELUsingAArch32(EL3) then
        return EL3;
    elsif IsInHost() then
        return EL2;
    else
        return EL1;
```

RETIRED

RETIRED

Part F

Appendixes

RETIRED

Appendix F1

Notes on Using Debug and Performance Monitors

This appendix describes aspects of the use of Debug, including the Performance Monitors, that are affected by ARMv8.1 or by the changes to the Performance Monitors introduced with ARMv8.1. It contains the following sections:

- [Self-hosted debug on page F1-952.](#)
- [External debug on page F1-953.](#)
- [Performance Monitors on page F1-954.](#)

F1.1 Self-hosted debug

Software uses the MDSCR_EL1.KDE, MDCR_EL2.TDE, and HCR_EL2.TGE controls to configure the current debug domain, and the DBGBCR<n>_EL1.{SSC, HMC, PMC} and the DBGWCR<n>_EL1.{SSC, HMC, PAC} fields to configure the Exception levels at which breakpoints and watchpoints trigger exceptions. This table shows the use of those controls.

What is being debugged	Exception level	Who is the debugger	KDE	TGE	TDE	Value programmed into {SSC, HMC, PMC}	Access to BCR/WCR
Guest App	EL0	Guest OS	0	0	0	{0x, 0, 10}	EL1 and EL2
Guest OS	EL1	Guest OS	1	0	0	{0x, 0, 01} ^a	EL1 and EL2
Guest App	EL0	Host OS	0	0	1	{0x, 0, 10}	EL2 only
Guest OS	EL1	Host OS	0	0	1	{0x, 0, 01}	EL2 only
Host App	EL0	Host OS	0	1	X	{0x, 0, 10}	EL2 only
Host OS	EL2	Host OS	1	1	X	{11, 1, 00}	EL2 only
Host OS and Hypervisor	EL2	Host OS	1	X	1	{11, 1, 00}	EL2 only

a. See *Use of the case {SSC, HMC, PMC} == {0x, 0, 01}*.

F1.1.1 Use of the case {SSC, HMC, PMC} == {0x, 0, 01}

The configuration {00, 1, 01} can be used in cases where:

- Guest OS is being debugged by Guest, that is, MDCR_EL2.TDE==0 and HCR_EL2.TGE==0. These breakpoints and watchpoints will not generate an exception at EL2 when MDCR_EL2.TDE==0 and HCR_EL2.TGE==0.
- Host OS is being debugged by Host, but only when HCR_EL2.TGE==1.

This means the code for these two cases can be identical. The values shown allow the Host OS, and Host OS and Hypervisor cases to be the same.

If debug exceptions are not being used from EL2, the Host OS uses MDSCE_EL2.{MDE, SS} as a quick path to disable all breakpoints, watchpoints, and software step when switching from a Guest using self-hosted debug into a Host App. This prevents breakpoints and watchpoints programmed by the Guest from generating exceptions in the Host App.

F1.2 External debug

An external debugger uses the DBGBCR<n>_EL1.{SSC, HMC, PMC} and the DBGWCR<n>_EL1.{SSC, HMC, PAC} fields to configure the Exception levels at which breakpoints and watchpoints trigger entry to Debug state, and the context matching breakpoints described in [Self-hosted debug on page F1-952](#) to link that to a particular Host or Guest Operating System or App. This table shows the use of those controls.

What is being debugged	Exception level	Value programmed into {SSC, HMC, PMC}	Linked to Context matching type	BT value
Guest App	EL0	{01, 0, 10}	Linked Full Context ID match	1111
Guest OS	EL1	{01, 0, 01}	Linked CONTEXTIDR_EL2 match	1101
Host App	EL0	{01, 0, 10}	Linked CONTEXTIDR_EL2 match	1101
Host OS and Hypervisor	EL2	{11, 1, 00}	Unlinked	-

F1.3 Performance Monitors

Software uses the MDCR_EL2.TPM control to configure the current profiling domain, and the PMEVTYPER<n>_EL1.{P, U, NSH} fields to configure the Exception levels at which the counters count events. This table shows the use of those controls.

What is being profiled	Exception level	Profiler	TGE	HPMD	TPM	Value programmed into {P, U, NSH}	General access to PM* registers
Guest App	EL0	Guest OS	0	1	0	{1, 0, 0}	EL1 and EL2 ^a
Guest OS	EL1	Guest OS	0	1	0	{0, 1, 0}	EL1 and EL2 ^a
Guest OS and App	EL1, EL0	Guest OS	0	1	0	{0, 0, 0}	EL1 and EL2 ^a
Guest App	EL0	Host OS	0	x	1	{1, 0, 0}	EL2 only
Guest OS	EL1	Host OS	0	x	1	{0, 1, 0}	EL2 only
Guest OS and App	EL1, EL0	Host OS	0	x	0	{0, 0, 0}	EL2 only
Host App	EL0	Host OS	1	x	x	{1, 0, 0}	EL2 only ^b
Host OS and Hypervisor	EL2	Host OS	x	0	1	{1, 1, 1}	EL2 only
Guest and Host OS, Hypervisor, and App	EL2, EL1, EL0	Host OS	x	0	1	{0, 0, 1}	EL2 only

a. EL1 can also grant access to the Performance Monitors to EL0 using PMUSERENR_EL0.

b. If TPM = 0, then EL2 can also grant access to the performance monitors to EL0 using PMUSERENR_EL0.

If profiling is not being used at EL2, the Host Operating System uses PMCR_EL1.E as a quick path to disable all counters when switching from a Guest using profiling into a Host App. This prevents counters programmed by the Guest from counting events in the Host App.

F1.3.1 Partitioning counters

A Host can also use MDCR_EL2.HPMN to partition the counters and present only a subset to a Guest. In this case, the Guest can program these counters, but not those programmed by the Host.

If the Host also sets MDCR_EL2.HPMD, then the counters programmed by the Guest will not count at EL2. However, they will count inside any Host App. The counters reserved by the Host will count at EL2, if the NSH bit is set.