Arm® Architecture Reference Manual Supplement

Supplement

Armv8, for Armv8-R AArch64 architecture profile

arm

Arm Architecture Reference Manual Supplement
Armv8, for Armv8-R AArch64 architecture profile

Copyright © 2019-2020 Arm Limited or its affiliates. All rights reserved.

Release Information

The following changes have been made to this document.

<div align="right">Release history</div>

Date	Issue	Confidentiality	Change
14 January 2020	A.a	Confidential	Beta release
19 June 2020	A.b	Confidential	Second beta release
7 September 2020	A.c	Non-Confidential	Initial EAC release of the PMSA architecture, first Beta release of the VMSA architecture

---
Note

- The term Arm can refer to versions of the Arm architecture, for example Armv8 refers to version 8 of the Arm architecture. The context makes it clear when the term is used in this way.

- This document describes only the Armv8-R AArch64 architecture profile. For the behaviors required by the Armv8-A architecture, see the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

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

This manual covers two memory system architectures, Protected Memory System Architecture (PMSAv8-64) and Virtual Memory System Architecture (VMSAv8-64). The information related to PMSAv8-64 as described in Chapter C1 *Protected Memory System Architecture* is at EAC quality. EAC quality means that all features of the specification are described in the manual.

The information related to VMSAv8-64 as described in Chapter D1 *Virtual Memory System Architecture* is at Beta quality. Beta quality means that:

- All major features of the specification are described in the manual, some details might be missing.
- Information can be used for software development at risk.
- Information should not be used for hardware development.

Web Address

`http://www.arm.com`

Limitations of this issue

This issue of the *Arm® Architecture Reference Manual Supplement Armv8, for Armv8-R AArch64 architecture profile* contains many improvements and corrections. Validation of this document has identified the following issues that Arm will address in future issues:

- The references to LDLAR, LDLARH, and SMC instructions are present in register descriptions.
- In Part I *Architectural Pseudocode*:
  — The functions that address both AArch32 and AArch64 functionality might contain cases, comments, or references that apply to only AArch32 state, EL3 Exception level, Monitor mode, Non-secure state, or other features that are not supported in Armv8-R AArch64, and are therefore not applicable to the Armv8-R AArch64 architecture.
  — Some functions and comments might contain information that is related to the short-descriptor format that is not applicable to the Armv8-R AArch64 architecture.
- Assertions that are not applicable to Armv8-R AArch64 might be present.
- Enumerations might contain values that are not applicable to Armv8-R AArch64.
- Tests might contain clauses that always return TRUE or FALSE in AArch64 state and there could be potentially redundant tests in the Armv8-R AArch64 architecture. For example, in Armv8-R AArch64:
  — `UsingAArch32()` always returns FALSE.
  — `IsSecure` always returns TRUE.

Contents
Arm Architecture Reference Manual Supplement Armv8, for Armv8-R AArch64 architecture profile

Part H   Armv8-R AArch64 External Debug Registers

Part I   Architectural Pseudocode

Glossary

Preface

This preface introduces the *Arm® Architecture Reference Manual Supplement Armv8, for Armv8-R AArch64 architecture profile*. It contains the following sections:

About this supplement

This supplement describes the changes that are introduced by the Armv8-R AArch64 architecture. For a summary of these changes, see *The Armv8-R AArch64 architecture profile* on page A1-22.

The 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-R AArch64 architecture.

This manual is organized into parts as described in *Using this book* on page xi.

Using this book

The purpose of this book is to describe the changes that are introduced by the Armv8-R AArch64 architecture. It describes the Armv8-R AArch64 profile in terms of how it differs from the Armv8-A AArch64 profile.

This book is a supplement to the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*, (ARM DDI 0487), and is intended to be used with it. There might be inconsistencies between this supplement and the Armv8-A Architecture Reference Manual due to some late-breaking changes. Therefore, the Armv8-A Architecture Reference Manual is the definitive source of information about Armv8-A.

It is assumed that the reader is familiar with the Armv8-A and Armv8-R architectures.

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

Part A, Introduction and Architecture Overview

Chapter A1 *Architecture Overview*

Provides an introduction to the Armv8 architecture, the Armv8-R AArch64 architecture profile, and the architecture extensions supported in Armv8-R AArch64.

Part B, Differences between the Armv8-A AArch64 and the Armv8-R AArch64 Profiles

Chapter B1 *Differences between the Armv8-A AArch64 and the Armv8-R AArch64 Profiles*

Describes the system level and application level architectural differences between the Armv8-A AArch64 and the Armv8-R AArch64 profiles.

Part C, Protected Memory System Architecture

Chapter C1 *Protected Memory System Architecture*

Read this for a system level view of the Armv8-R AArch64 Protected Memory System Architecture.

Part D, Virtual Memory System Architecture

Chapter D1 *Virtual Memory System Architecture*

Read this for a system level view of the Armv8-R AArch64 Virtual Memory System Architecture.

Part E, A64 Instruction Set for Armv8-R AArch64

Chapter E1 *A64 Instruction Set for Armv8-R AArch64*

Read this for descriptions of the A64 instructions that are added or affected by the Armv8-R AArch64 architecture profile.

Part F, The A64 System Instructions

Chapter F1 *The A64 System Instructions*

Read this for the descriptions of A64 System instructions.

Part G, Armv8-R AArch64 System Registers

Part G describes the System registers for Armv8-R AArch64. It contains the following chapters:

Chapter G1 *System Registers in a PMSA Implementation*

Read this for descriptions of Armv8-R AArch64 System registers in a Protected Memory System Architecture (PMSAv8-64) implementation.

Chapter G2 *System Registers in a VMSA Implementation*

> Read this for descriptions of Armv8-R AArch64 System registers in a Virtual Memory System Architecture (VMSAv8-64) implementation.

Part H, Armv8-R AArch64 External Debug Registers

Chapter H1 *External Debug Registers Descriptions*

> Read this for descriptions of the External debug registers that are added or affected by the Armv8-R AArch64 architecture profile.

Part I, Architectural Pseudocode

Chapter I1 *Armv8-R AArch64 Pseudocode*

> Contains pseudocode that describes various features of the Armv8-R AArch64 architecture profile.

Glossary

Defines terms used in this document that have a specialized meaning.

———— **Note** ————

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.

Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*.
- *Signals*.
- *Numbers*.
- *Pseudocode descriptions*.

Typographic conventions

The following table describes the typographic conventions:

Typographic conventions

Style	Purpose
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 included in the *Glossary* in the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.
Colored text	Indicates a link. This can be:
	• A URL, for example https://developer.arm.com.
	• A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Pseudocode descriptions*.
	• 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 A1.

Signals

In general this specification does not define processor signals, but it does include some signal examples and recommendations.

The signal conventions are:

Signal level   The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:
- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n   At the start or end of a signal name denotes an active-LOW signal.

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.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality.

Additional reading

This section lists relevant publications from Arm and third parties.

See Developer, https://developer.arm.com, for access to Arm documentation.

Arm publications

- *Arm® Architecture Reference Manual, Armv7-A and Armv7-R edition* (ARM DDI 0406).
- *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile* (ARM DDI 0487).
- *Arm® CoreSight™ Architecture Specification v3.0* (ARM IHI 0029).
- *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.0 to ETMv4.5* (ARM IHI 0064).
- *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

Other publications

- JEDEC Solid State Technology Association, *Standard Manufacturer's Identification Code*, JEP106.

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, *Arm® Architecture Reference Manual Supplement Armv8, for Armv8-R AArch64 architecture profile*.
*   The number, ARM DDI 0600A.c.
*   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.

Part A

Introduction and Architecture Overview

Chapter A1
Architecture Overview

This chapter introduces the Armv8 architecture, the architecture profiles it defines, and the Armv8-R AArch64 profile that this manual defines. It contains the following sections:

A1.1   About the Armv8 architecture

The Arm architecture that this Architecture Reference Manual describes, defines the behavior of an abstract machine, referred to as a *processing element* (PE). The implementations that are compliant with the Arm architecture must conform to the described behavior of the PE. This manual does not describe how to build an implementation of the PE, nor does it limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the Arm architecture must be the same as a simple sequential execution of the program on the PE. This programmer-visible behavior does not include the execution time of the program.

The Arm Architecture Reference Manual also describes rules for software to use the PE.

The Arm architecture includes definitions of:

*   An associated debug architecture.

*   Associated trace architectures, which define trace macrocells that implementers can implement with the associated processor hardware.

The Arm architecture is a *Reduced Instruction Set Computer* (RISC) architecture with the following RISC architecture features:

*   A large uniform register file.

*   A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.

*   Simple addressing modes, with all *load/store* addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system. This document defines the Armv8-R AArch64 architecture profile. See *The Armv8-R AArch64 architecture profile* on page A1-22 for more information.

The Arm architecture supports implementations across a wide range of performance points. Implementation size, performance, and low power consumption are key attributes of the Arm architecture.

See *Conventions* on page xiii for information about conventions used in this manual, including the use of SMALL CAPS for particular terms that have Arm-specific meanings that are defined in the *Glossary*.

A1.1.1   See also

In the Arm Architecture Reference Manual

*   Introduction to the Armv8 Architecture.
*   Armv8 architectural concepts.

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Eight major versions of the architecture have been defined to date, denoted by the version numbers 1 to 8. Of these, the first three versions are now obsolete.

Armv8 defines three architecture profiles:

A   Application profile:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

- Supports the A32, T32, and A64 instruction sets.

R   Real-time profile:

- Supports the AArch64 or AArch32 Execution states.

- Supports A64, or A32 and T32 instruction sets.

- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).

- Supports a VMSA based on an MMU.

M   Microcontroller profile:

- Implements a programmers' model that is designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.

- Supports a PMSA based on an MPU.

- Supports a variant of the T32 instruction set.

For more information, see *Introduction to the Armv8 Architecture* chapter of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

A1.3 The Armv8-R AArch64 architecture profile

The Armv8-R AArch64 architecture profile is described in terms of Armv8-A Exception levels. Armv8-R AArch64 implementations support EL0, EL1, and EL2 Exception levels.

The main features of the Armv8-R AArch64 profile are:

- Support for one Execution state, AArch64.
- No EL3 Exception level. Secure monitor is not supported in the Armv8-R AArch64 profile.
- A PMSA that defines memory ordering and memory management in 64-bit address space and provides:
  — A model for defining protection regions at EL1 and EL2 using two 64-bit registers to specify a base address and a limit address.
  — A minimum protection region size of 64 bytes.
  — No support for overlapping protection regions.
- The PE is always in Secure state.
- A programmers' model and its interfaces to AArch64 registers with EL1 and EL2 PMSA registers that control most PE and memory system features, and provide status information.
- Support for Advanced SIMD and floating-point instructions.
- The Armv8-R AArch64 virtualization model, which provides:
  — Support for the EL2 Exception level.
  — A second MPU that provides stage 1 memory protection for memory accesses from EL2 and provides stage 2 memory protection for accesses from EL1 and EL0. These protection stages act as address translation regimes in the Armv8-R AArch64 profile.
- ETMv4.5 with Armv8-R AArch64 extension.

  The Arm architecture includes definitions of associated trace architectures, which define trace macrocells that implementers can implement with the associated processor hardware. For more information, see *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.0 to ETMv4.5* (ARM IHI 0064).
- Support for GICv3 or GICv4. For more information, see *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).
- The Armv8 AArch64 Debug architecture that provides software access to debug features.
- Optional support for Virtual Memory System Architecture (VMSAv8-64) extension. VMSAv8-64 provides virtual memory addressing support.

A1.3.1 See also

In the Arm Architecture Reference Manual

- The AArch64 Application Level Programmers' Model.
- The AArch64 Application Level Memory Model.
- The AArch64 System Level Programmers' Model.
- AArch64 Self-hosted Debug.
- AArch64 Self-hosted Trace.
- The AArch64 System Level Memory Model.
- The AArch64 Virtual Memory System Architecture.

A1.4 Architecture extensions

An implementation of the Armv8-R AArch64 architecture is based on the Armv8.4-A architecture. Table A1-1 lists the features supported by the Armv8-R AArch64 architecture.

Table A1-1 Armv8-A features supported in Armv8-R AArch64

Feature	Description
FEAT_SSBS	Speculative Store Bypass Safe
FEAT_CSV2	Cache Speculation Variant 2
FEAT_CSV3	Cache Speculation Variant 3
FEAT_SB	Speculation Barrier
FEAT_SPECRES	Speculation restriction instructions
FEAT_SHA1	Advanced SIMD SHA1 instructions
FEAT_SHA256	Advanced SIMD SHA256 instructions
FEAT_AES	Advanced SIMD AES instructions
FEAT_PMULL	Advanced SIMD PMULL instructions
FEAT_PCSRv8	PC Sample-based Profiling Extension
FEAT_DGH	Data Gathering Hint
FEAT_LSE	Large System Extensions
FEAT_RDM	Advanced SIMD rounding double multiply accumulate instructions
FEAT_PAN	Privileged access never
FEAT_VMID16	16-bit VMID
FEAT_PMUv3p1	PMU Extensions v3.1
FEAT_XNX	Translation table stage 2 Unprivileged Execute-never
FEAT_UAO	Unprivileged Access Override control
FEAT_PAN2	AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN
FEAT_DPB	DC CVAP instruction
FEAT_Debugv8p2	Debug v8.2
FEAT_ASMv8p2	Armv8.2 changes to the A64 ISA
FEAT_IESB	Implicit Error Synchronization event
FEAT_DPB2	DC CVADP instruction
FEAT_FP16	Half-precision floating-point data processing
FEAT_LVA	Large VA support
FEAT_LPA	Large PA and IPA support
FEAT_VPIPT	VMID-aware PIPT instruction cache

Table A1-1 Armv8-A features supported in Armv8-R AArch64 (continued)

Feature	Description
FEAT_PCSRv8p2	PC Sample-based profiling
FEAT_DotProd	Advanced SIMD dot product instructions
FEAT_SHA3	Advanced SIMD SHA3 instructions
FEAT_SHA512	Advanced SIMD SHA512 instructions
FEAT_SM3	Advanced SIMD SM3 instructions
FEAT_SM4	Advanced SIMD SM4 instructions
FEAT_FHM	Floating-point half-precision multiplication instructions
FEAT_PAuth	Pointer authentication
FEAT_JSCVT	JavaScript conversion instructions
FEAT_LRCPC	Load-acquire RCpc instructions
FEAT_FCMA	Floating-point complex number instructions
FEAT_DoPD	Debug over Powerdown
FEAT_CCIDX	Extended cache index
FEAT_PAuth2	Enhancements to pointer authentication
FEAT_FPAC	Faulting on AUT* instructions
FEAT_SEL2	Secure EL2
FEAT_S2FWB	Stage 2 forced Write-Back
FEAT_DIT	Data Independent Timing instructions
FEAT_IDST	ID space trap handling
FEAT_FlagM	Flag Manipulation instructions v2
FEAT_LSE2	Large System Extensions v2
FEAT_LRCPC2	Load-acquire RCpc instructions v2
FEAT_TLBIOS	TLB invalidate instructions in Outer Shareable domain
FEAT_TLBIRANGE	TLB invalidate range instructions
FEAT_CNTSC	Generic Counter Scaling
FEAT_RASv1p1	RAS Extension v1.1
FEAT_Debugv8p4	Debug v8.4
FEAT_PMUv3p4	PMU Extensions v3.4
FEAT_TRF	Self-hosted Trace Extensions

The Armv8-R AArch64 architecture supports concurrent modification and execution of instructions as defined by the Armv8-A architecture. FEAT_IDST feature is extended to include MPUIR_EL1 register.

For the architectural features supported by Armv8-R AArch64, whether a feature is mandatory or optional depends on whether the feature is mandatory or optional in the Armv8.4-A architecture.

In a PMSAv8-64 only implementation, the FEAT_TLBIOS and FEAT_TLBIRANGE features are optional.

A1.4.1   See also

In the Arm Architecture Reference Manual
- Armv8-A Architecture Extensions.
- The Armv8.1 architecture extension.
- The Armv8.2 architecture extension.
- The Armv8.3 architecture extension.
- The Armv8.4 architecture extension.

A1.5 Supported extensions in Armv8-R AArch64

A1.5.1 Advanced SIMD and Floating-point extensions

The support for Advanced SIMD and floating-point instructions must conform to the Armv8-A AArch64 specifications.

A1.5.2 See also

In the Arm Architecture Reference Manual
- Advanced SIMD and floating-point support.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.

Part B

Differences between the Armv8-A AArch64 and the Armv8-R AArch64 Profiles

Chapter B1
Differences between the Armv8-A AArch64 and the Armv8-R AArch64 Profiles

This chapter describes the system level and application level architectural differences between Armv8-R AArch64 and Armv8-A AArch64. It contains the following sections:

- *Differences from the Armv8-A AArch64 application level architecture* on page B1-30.
- *Differences from the Armv8-A AArch64 system level architecture* on page B1-31.

B1.1 Differences from the Armv8-A AArch64 application level architecture

B1.1.1 Differences from the Armv8-A AArch64 application level programmers' model

The Armv8-R AArch64 application level programmers' model differs from the Armv8-A AArch64 profile in the following ways:

- Armv8-R AArch64 supports only a single Security state, Secure.
- EL2 is mandatory.
- EL3 is not supported.
- Armv8-R AArch64 supports the A64 ISA instruction set with some modifications.

See *The AArch64 Application Level Programmers' Model* chapter of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

B1.1.2 Differences from the Armv8-A AArch64 application level memory model

Armv8-R AArch64 redefines `Data Memory Barrier` and `Data Synchronization Barrier`, and adds an instruction, `Data Full Barrier`.

B1.1.3 See also

In the Arm Architecture Reference Manual

- Memory barriers.
- Data Memory Barrier (DMB).
- Data Synchronization Barrier (DSB).
- Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

B1.2 Differences from the Armv8-A AArch64 system level architecture

B1.2.1 Differences from the Armv8-A AArch64 system level programmers' model

Virtualization

Armv8-R AArch64 provides a PMSA-based virtualization model.

Generic Interrupt Controller

Armv8-R AArch64 supports GICv3 or GICv4. The GIC architecture is defined by the *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

B1.2.2 Differences from the Armv8-A AArch64 system level memory model

Address space

Armv8-R AArch64 uses flat mapping from the *virtual address* (VA), used by the PE, to the *physical address* (PA), and can support address bits up to 52 if FEAT_LPA is enabled, otherwise 48 bits.

Address translation

In PMSAv8-64 address translation flat-maps the VA to the PA, and determines the access permissions and memory attributes of the target PA.

System register support for IMPLEMENTATION DEFINED memory features

The type, presence, and accessibility of *Tightly Coupled Memory* to EL1 and EL0, or to EL2, is IMPLEMENTATION DEFINED.

B1.2.3 Protected Memory System Architecture, PMSAv8-64

The Armv8-R AArch64 implementation supports the Protected Memory System Architecture (PMSAv8-64) at EL1 and EL2. See Chapter C1 *Protected Memory System Architecture*.

B1.2.4 See also

In the Arm Architecture Reference Manual
- The Arm Generic Interrupt Controller System registers.
- About the GIC System registers.
- Address generation.
- Address space.
- Address size configuration.
- Address translation instructions.
- A64 System instructions for address translation.

Part C

Armv8-R AArch64 Protected Memory System Architecture

Chapter C1
Protected Memory System Architecture

This chapter provides a system-level view of the Protected Memory System Architecture for any implementation that is compliant with the Armv8-R AArch64 architecture. It contains the following sections:

C1.1 About the Protected Memory System Architecture

The Armv8-R AArch64 implementation supports the Protected Memory System Architecture (PMSAv8-64) at EL1 and EL2. The PMSAv8-64 is based on MPUs that provide a memory protection scheme by defining protection regions in the address space.

The PMSAv8-64:

- Supports a unified memory protection scheme where an MPU manages instruction and data access. It does not provide separate instruction protection regions and data protection regions in the address map.
- Defines MPU faults that are consistent with VMSAv8-64 fault definitions and reuses IFSC and DFSC fault encodings.
- Does not support virtual addressing and flat maps input address to output address.

For general information about the Arm memory model, see *The AArch64 Application Level Memory Model* and *The AArch64 System Level Memory Model* chapters of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

C1.1.1 Protection regions

An MPU defines protection regions in the address map. A protection region is a contiguous memory region for which the MPU defines the memory attributes and the access permissions.

Protection regions:

- Are defined by a pair of registers, a Base Address Register, and a Limit Address Register, see *Memory Protection Unit* on page C1-37.
- Have a minimum size of 64 bytes.
- Have a maximum size of the entire address map.
- Must not overlap.

The definition of a protection region specifies the start and the end of the region, the access permissions, and the memory attributes for the region.

C1.1.2 Address range

The maximum supported address bit size is 48, or 52 if FEAT_LPA is enabled. A PE can choose to implement a smaller PA range and the actual implemented physical address range is provided by the ID_AA64MMFR0_EL1.PARange field. Any access to physical memory address outside the address range results in a memory fault.

C1.2   Memory Protection Unit

An MPU checks whether the address used by a memory access matches a defined protection region. The MPU uses a Base Address Register and a Limit Address Register to define a protection region and its associated access permissions and memory attributes. The minimum supported size of a protection region is 64-bytes.

The PMSAv8-64 defines two MPUs:

EL1 MPU

> The EL1 MPU can be configured from EL1 or EL2. The EL1 MPU controls the stage 1 of the Secure EL1&0 translation regime that defines the protection regions for accesses from EL1 and EL0. The PMSAv8-64 uses SCTLR_EL1.M to enable and disable the EL1 MPU. The EL1 MPU also supports a Background region, controlled by SCTLR_EL1.BR.

EL2 MPU

> The EL2 MPU can be configured only from EL2. The EL2 MPU controls:
> * Stage 1 of the Secure EL2 translation regime that defines the protection regions for accesses from EL2.
> * Stage 2 of the Secure EL1 &0 translation regime that defines the protection regions for accesses from EL1 and EL0.
>
> The PMSAv8-64 uses SCTLR_EL2.M to enable and disable the EL2 MPU. The EL2 MPU also supports a Background region, controlled by SCTLR_EL2.BR.
>
> ───── **Note** ─────
>
> When HCR_EL2.VM is 1 and SCTLR_EL2.M is 1, then EL2 MPU modifies the access permissions and memory attributes that are assigned by the EL1 MPU.

See *Protection region attributes and access permissions* on page C1-49. PMSAv8-64 supports a default memory map as a Background region for memory region checks at both EL1 and EL2. See *Default memory map* on page C1-39.

C1.2.1   MPU Default Cacheability

The PMSAv8-64 supports Default Cacheability for the stage 1 of the Secure EL1&0 translation regime access and follows the same rule as Armv8-A.

For more information, see chapter *AArch64 System Register Descriptions* of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

C1.3   Address translation regimes

In PMSAv8-64:

- Address translation describes the process of flat mapping the VA used by the PE, to the PA accessed in the memory system, and determining the access permissions and memory attributes of the target PA.

- A translation regime maps a VA to a PA using one or two stages of address translation to assign the access permissions and memory attributes of the target PA. When two translation stages are used, the intermediate address is treated as an *intermediate physical address* (IPA).

The Armv8-R AArch64 architecture supports two translation regimes:

- Secure EL1&0 translation regime.
- Secure EL2 translation regime.

Secure EL1&0 translation regime

The Secure EL1&0 translation regime assigns the access permissions and memory attributes for any access from EL1 or EL0.

This translation regime has one or two stages of translation:

- All accesses from EL1 or EL0 are translated by the EL1 MPU. This translation is a stage 1 translation.

- When the value of HCR_EL2.VM is 1 and SCTLR_EL2.M is 1, the accesses are further translated by the EL2 MPU. This translation is a stage 2 translation, and can modify the access permissions and memory attributes that are assigned by the stage 1 translation.

For the EL1&0 stage 1 translation, an ADDRESS is in the protection region *n* if and only if:

PRBAR<n>_EL1.BASE:'000000' <= ADDRESS <= PRLAR<n>_EL1.LIMIT:'111111'.

Secure EL2 translation regime

The Secure EL2 translation regime assigns the access permissions and memory attributes for any access from EL2.

This translation regime has a single stage of translation, stage 1, that is performed by the EL2 MPU.

For the EL2 stage 1 translation, an ADDRESS is in the protection region *n* if and only if:

PRBAR<n>_EL2.BASE:'000000' <= ADDRESS <= PRLAR<n>_EL2.LIMIT:'111111'.

The attributes for a protection region are defined by the combination of:

- The values that are programmed into the Base Address Registers and Limit Address Registers. The registers are PRBAR_EL1, PRLAR_EL1, PRBAR_EL2, and PRLAR_EL2.

- A Memory Attributes Indirection register that is indexed by the values MAIR_EL1 and MAIR_EL2.

C1.4 Default memory map

For PMSAv8-64, the Background region is enabled and disabled using SCTLR_ELx.BR. If the Background region is enabled, then the MPU uses the default memory map as the Background region for generating the memory attributes when MPU is disabled.

The default memory map of the Armv8-R AArch64 architecture is IMPLEMENTATION DEFINED. Therefore, the Armv8-R AArch64 architecture defines only the condition to access the default memory map, but not the memory map itself. The memory attributes, access permissions, and Security state of the memory regions in the default memory map are also IMPLEMENTATION DEFINED.

Any access outside the implemented physical address range in the default memory map results in a fault.

If the IMPLEMENTATION DEFINED default memory map is discontinuous, then the implementation must also define a generic permission and attribute to be used for memory access to all memory regions that are not covered by the default memory map. However, an implementation can also select a default memory map so that the accesses to these discontinuous memory regions, where no memory attributes are allocated, always result in memory faults.

――― **Note** ―――

The default memory map is same for EL1 and EL2 MPUs.

C1.5　Armv8-A memory view

The PMSAv8-64 uses same controls as VMSAv8-64 to enable and disable translation stages.

If the MPU and the Background region are not enabled for stage 1 translation stage, then PMSAv8-64 uses the same memory attributes as defined by VMSAv8-64 when stage 1 translation is disabled.

C1.6   MPU memory translations and faults

This section provides information on memory attributes and the MPU faults for the stage 1 Secure EL1&0, stage 1 Secure EL2, and the stage 2 Secure EL1&0 translation regimes.

For the EL1 MPU memory translations, this section describes the following:

*   Stage 1 EL1&0 memory attributes.
*   Stage 1 MPU faults for EL1 access.
*   Stage 1 MPU faults for EL0 access.
*   EL1 MPU fault types.
*   MPU fault check sequence for the stage 1 Secure EL1&0 translation.

For the EL2 MPU memory translations, this section describes the following:

*   Stage 1 EL2 memory attributes.
*   Stage 1 MPU faults for EL2 access.
*   Stage 2 EL1&0 memory attributes.
*   Stage 2 EL1&0 MPU faults.
*   EL2 MPU fault types.
*   MPU fault check sequence for the stage 1 Secure EL2 translation.
*   MPU fault check sequence for the stage 2 Secure EL1&0 translation.

C1.6.1   EL1 MPU memory translations

The EL1 MPU controls the stage 1 of the Secure EL1&0 translation regime. Based on the values of HCR_EL2.DC and SCTLR_EL1.{M, BR}, the stage 1 of the Secure EL1&0 translation regime can have the following configurations for memory attributes, as described in Table C1-1.

Table C1-1 Stage 1 EL1&0 memory attributes

HCR_EL2	SCTLR_EL1		MPU hit	Memory attribute
DC	M	BR		
1	x	x	-	Default Cacheability
0	0	0	-	Armv8-A AArch64 memory view
0	0	1	-	Default memory map
0	1	0	No	Not applicable, MPU Fault
0	1	0	Yes	MPU memory map
0	1	1	No	Default memory map
0	1	1	Yes	MPU memory map

───── **Note** ─────

Armv8-A AArch64 memory view is the stage 1 memory attribute defined by the Armv8-A architecture for accessing a memory location when stage 1 address translation is disabled (SCTLR_ELx.M = 0).

Table C1-2 lists the configurations for the stage 1 MPU faults for EL1 access.

Table C1-2 Stage 1 MPU faults for EL1 access

HCR_EL2	SCTLR_EL1		MPU hit	MPU faults
DC	M	BR		
1	x	x	-	No Fault or Address size fault
0	0	0	-	No Fault or Address size fault
0	0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault
0	1	0	No	Translation fault
0	1	0	Yes	No Fault or Permission fault
0	1	1	No	No Fault, or Background region Translation fault, or Background region Permission fault
0	1	1	Yes	No Fault or Permission fault

Table C1-3 lists the configurations for the stage 1 MPU faults for EL0 access.

Table C1-3 Stage 1 MPU faults for EL0 access

HCR_EL2	SCTLR_EL1		MPU hit	MPU faults
DC	M	BR		
1	x	x	-	No Fault or Address size fault
0	0	0	-	No Fault or Address size fault
0	0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault
0	1	x	No	Translation fault
0	1	x	Yes	No Fault or Permission fault

———— **Note** ————

If HCR_EL2.{DC, TGE} is not {0, 0}, then the PE behaves as if the value of the SCTLR_EL1.BR is 0 for all purposes other than returning the value of a direct read of the field.

C1.6.2 EL1 MPU faults

Each EL1 MPU protection region is defined using the PRBAR_EL1 and PRLAR_EL1 registers. The MPU checks the input address with each protection region, and an address is considered to match a region if:

```
Address >= PRBAR_EL1.BASE:'000000' && Address <= PRLAR_EL1.LIMIT:'111111'
```

Based on MPU protection region checks, the EL1 MPU can raise the following responses as described in Table C1-4.

Table C1-4 EL1 MPU fault types

Protection region match	Permission	MPU response
No match	-	Translation fault
Multiple	-	Translation fault
Single	Denied	Permission fault
	Allowed	Valid

If the EL1 MPU is disabled and the input address is larger than the implemented PA size, then a level 0 address size fault is generated. If the EL1 MPU is enabled and the input address is larger than the implemented PA size, then a level 0 translation fault is generated. Permitted transactions are then presented to stage 2 permission checks by the EL2 MPU.

Depending on the configuration in the PMSAv8-64 registers, the memory attributes of an address can be defined by an MPU protection region, a Background region, or it may have Armv8-A AArch64 memory view.

C1.6.3 MPU fault check for the stage 1 Secure EL1&0 translation

Figure C1-1 shows the MPU fault check sequence for the stage 1 of the Secure EL1&0 translation regime.

Figure C1-1 MPU fault check sequence for the stage 1 Secure EL1&0 translation

C1.6.4 EL2 MPU memory translations

Based on the values of SCTLR_EL2.{M, BR}, the stage 1 of the Secure EL2 translation regime can have the following configurations for memory attributes, as described in Table C1-5.

Table C1-5 Stage 1 EL2 memory attributes

SCTLR_EL2		MPU hit	Memory attributes
M	BR		
0	0	-	Armv8-A AArch64 memory view
0	1	-	Default memory map
1	0	No	Not applicable. MPU Fault.
1	0	Yes	MPU memory map
1	1	No	Default memory map
1	1	Yes	MPU memory map

Table C1-6 lists the configurations for stage 1 MPU faults for EL2 access.

Table C1-6 Stage 1 MPU faults for EL2 access

SCTLR_EL2		MPU hit	MPU faults
M	BR		
0	0	-	No Fault or Address size fault
0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault
1	0	No	Translation fault
1	0	Yes	No Fault or Permission fault
1	1	No	No Fault, or Background region Translation fault, or Background region Permission fault
1	1	Yes	No Fault or Permission fault

Based on HCR_EL2.VM and SCTLR_EL2.{M, BR}, the stage 2 of the Secure EL1&0 translation regime can have the following configurations for memory attributes, as described in Table C1-7.

Table C1-7 Stage 2 EL1&0 memory attributes

HCR_EL2	SCTLR_EL2		MPU hit	Memory attribute
VM	M	BR		
0	x	x	-	Stage 2 translation disabled
1	0	0	-	CONSTRAINED UNPREDICTABLE
1	0	1	-	Default memory map
1	1	x	No	Not applicable. MPU Fault.
1	1	x	Yes	MPU memory map

Table C1-8 lists the configurations for MPU faults for the stage 2 of the Secure EL1&0 translation regime.

Table C1-8 Stage 2 EL1&0 MPU faults

HCR_EL2	SCTLR_EL2		MPU hit	MPU faults
VM	M	BR		
0	x	x	-	No Fault
1	0	0	-	CONSTRAINED UNPREDICTABLE
1	0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault, or Translation fault due to Secure Check, if enabled.
1	1	x	No	Translation fault
1	1	x	Yes	No Fault, or Permission fault, or Translation fault due to Secure Check, if enabled.

There are no separate configurations for the protection regions for the stage 1 of the Secure EL2 and the stage 2 of the Secure EL1&0 translations. Memory accesses for both translations are controlled by the same MPU configuration registers, PRBAR_EL2 and PRLAR_EL2.

─── **Note** ───

In Armv8-A, there are separate translation table base registers for the stage 1 of the Secure EL2 and the stage 2 of the Secure EL1&0 translation regimes.

C1.6.5   EL2 MPU faults

Each EL2 MPU protection region is defined using PRBAR_EL2 and PRLAR_EL2. The MPU checks the input address with each protection region and an address is considered to match a region if:

```
Address >= PRBAR_EL2.BASE:'000000' && Address <= PRLAR_EL2.LIMIT:'111111'
```

Based on MPU protection region checks, the EL2 MPU can raise the following responses as described in Table C1-9.

Table C1-9 EL2 MPU fault types

Protection region match	Permission	MPU response
No match	-	Translation fault
Multiple	-	Translation fault
Single	Denied	Permission fault
Single	Allowed	Valid

If the EL2 MPU is disabled for the stage 1 of the Secure EL2 translation regime, then any access to an address outside the implemented PA range raises a level 0 address size fault. If the EL2 MPU is enabled for the stage 1 of the Secure EL2 translation regime, and the input address is larger than the implemented PA range, then a level 0 translation fault is generated.

C1.6.6 MPU fault check for the stage 1 Secure EL2 translation

Figure C1-2 shows the MPU fault check sequence for the stage 1 of the Secure EL2 translation regime.
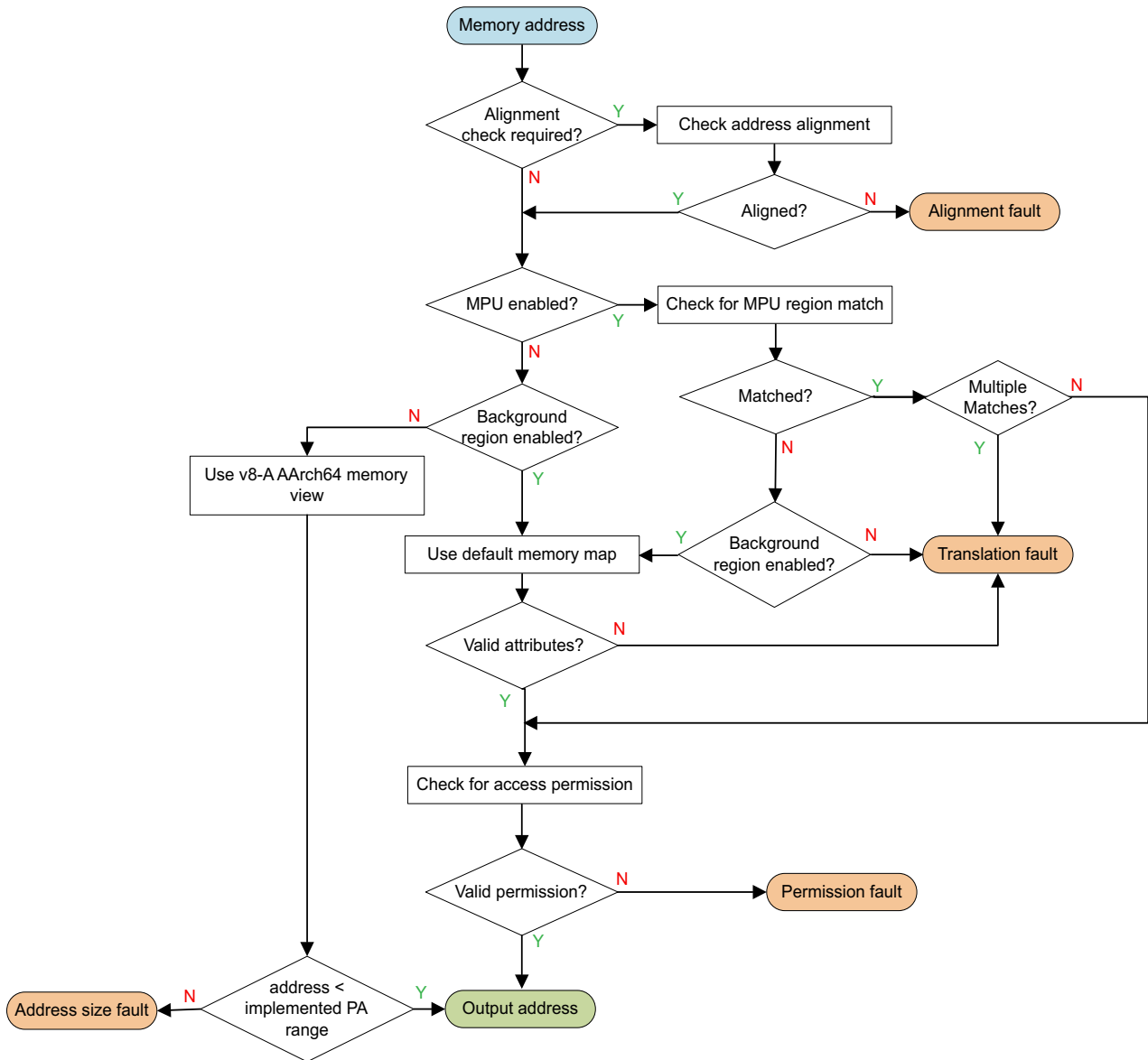
Figure C1-2 MPU fault check sequence for the stage 1 Secure EL2 translation

C1.6.7 MPU fault check for the stage 2 Secure EL1&0 translation

Figure C1-3 shows the MPU fault check sequence for the stage 2 of the Secure EL1&0 translation regime.
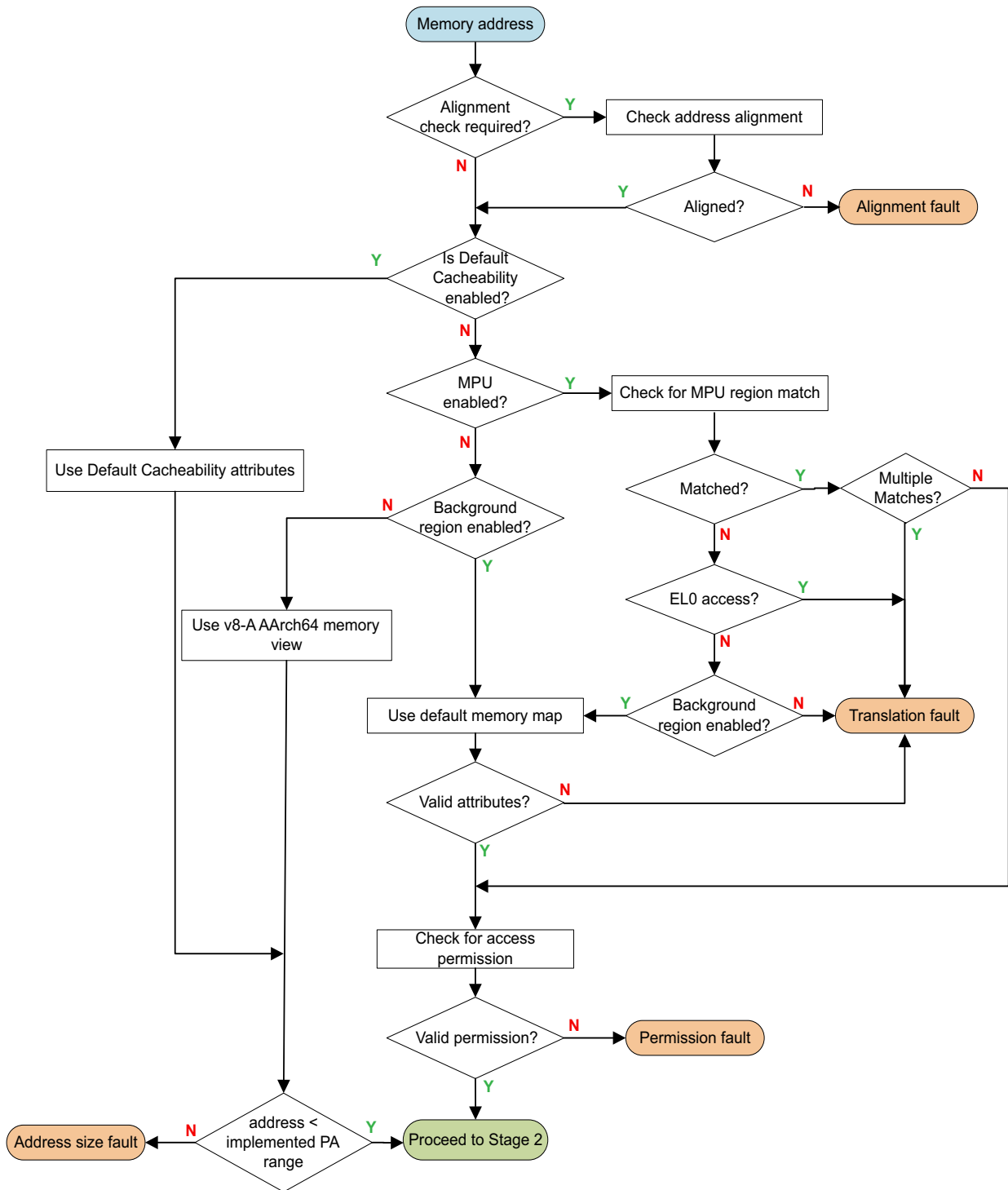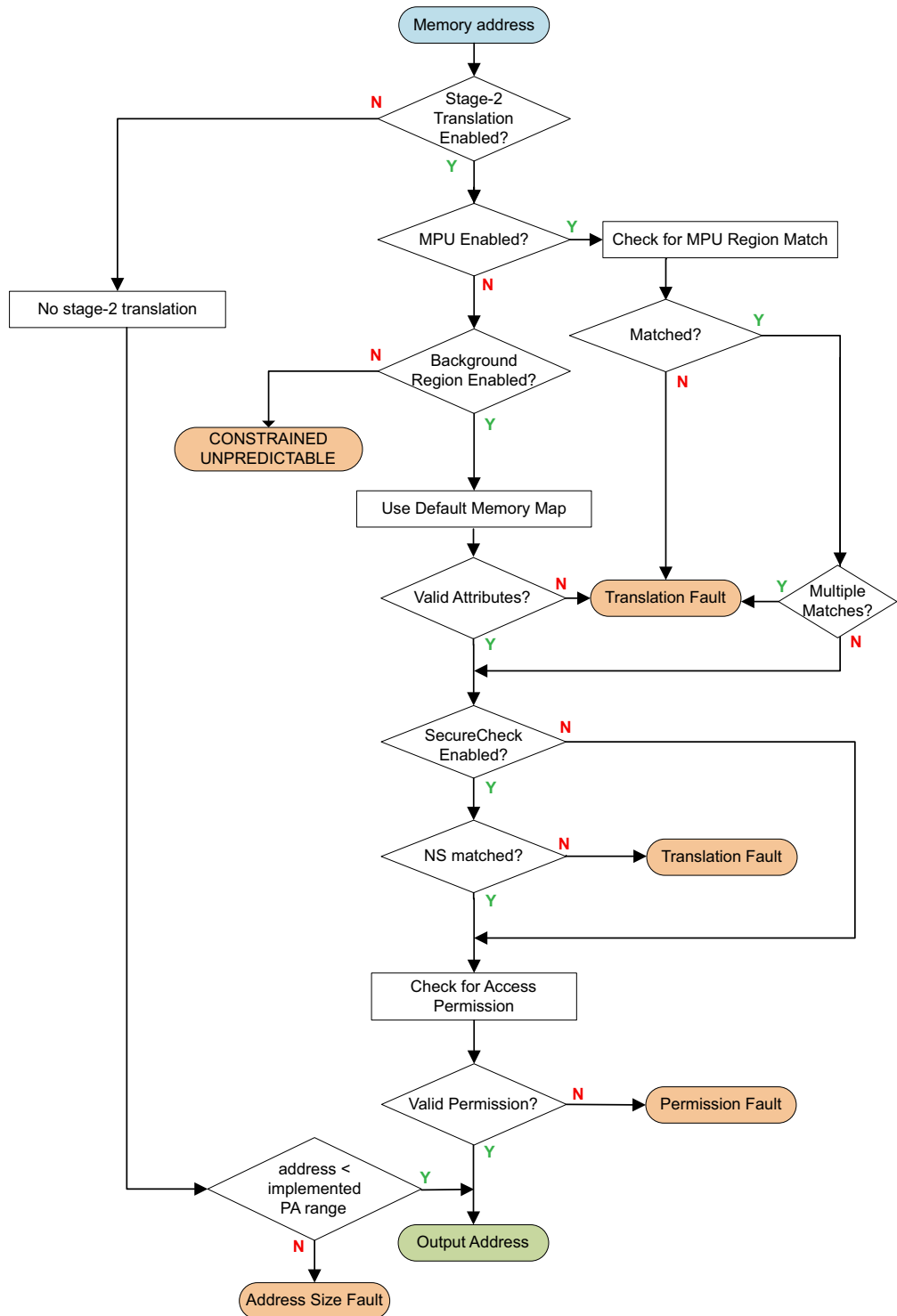

Figure C1-3 MPU fault check sequence for the stage 2 Secure EL1&0 translation

---

C1.7 Protection region attributes and access permissions

The protection region attribute fields control the Memory type, Cacheability, and Shareability attributes of the region. PMSAv8-64 uses the same memory types and memory attributes as VMSAv8-64 in Armv8-A.

The memory attributes and access permissions for a protection region are defined by:

- The PRBAR_EL1 and PRLAR_EL1 registers, or the PRBAR_EL2 and PRLAR_EL2 registers that define the protection region.

- The MAIR_EL1.Attr<*n*> or MAIR_EL2.Attr<*n*> field that is indexed by PRLAR_EL1.AttrIndx or PRLAR_EL2.AttrIndx, respectively.

See also *Memory attributes and access permission mappings* on page C1-51.

For the Secure EL1&0 translation regime, when HCR_EL2.VM is 1, the stage 1 memory attribute and access permission assignments are combined with the stage 2 assignments, as described in *Combining memory attributes and access permissions* on page C1-51.

C1.7.1 Protection region attributes

The PMSAv8-64 uses the same memory attributes defined by the VMSAv8-64, and the MPU enables configuration of the attributes for each protection region using MAIR_EL1 and MAIR_EL2. The PMSAv8-64 also enables configuration of each protection region to map to Secure or Non-secure address space using the NS bit in the EL1 and EL2 MPU configuration registers.

───── **Note** ─────

Writes to MPU registers are only guaranteed to be visible following a Context synchronization event and `DSB` operation.

───────────

If there are multiple protection regions allocated to the same coherency granule, then Armv8-R AArch64 follows the Armv8-A mismatched memory attributes rules to access any byte within that coherency granule. If the Security states of protection regions are different, then implementation must ensure that these regions are not allocated to the same coherency granule.

The PMSAv8-64 uses the same memory attributes defined by the VMSAv8-64 using MAIR_EL1 and MAIR_EL2 for the stage 1 EL1&0 and EL2 translations. For the stage 2 EL1&0 translations, memory attributes encoding in the MAIR_EL2 register is defined in Table C1-10.

Table C1-10 Meaning of Attr[7:4]

Attr [7:6]	Attr [5:4]	Memory type	Outer cache policy
00	00	Device memory	Not Applicable (NA)
00	!=00	Normal memory	Write-Through (WT)
01	00	Normal memory	Non Cacheable (NC)
01	!=00	Normal memory	Write-Back (WB)
10	xx	Normal memory	Write-Through (WT)
11	xx	Normal memory	Write-Back (WB)

When Attr[7:4] is 0b0000, Attr[3:0] defines the type of Device memory. In this case, Attr[1:0] != 0b00 gives UNPREDICTABLE behavior as defined by Armv8-A. Table C1-11 describes the meaning of Attr<3.0> when Attr[7:4] is 0b0000.

Table C1-11 Attr<3.0> Meaning when Attr[7:4] is 0b0000

Attr [3:2]	Attr [1:0]	Memory type
00	00	Device-nGnRnE
01	00	Device-nGnRE
10	00	Device-nGRE
11	00	Device-GRE

When Attr[7:4] is not 0b0000, Attr[3:0] defines the Inner Cache Policy, and Attr[3:0] = 0b0000 gives UNPREDICTABLE behavior as defined by Armv8-A. Table C1-12 describes the meaning of Attr<3.0> when Attr[7:4] is not 0b0000.

Table C1-12 Attr<3.0> Meaning when Attr[7:4] is not 0b0000

Attr [3:2]	Attr [1:0]	Memory type	Inner Cache Policy
00	!=00	Normal memory	Write-Through (WT)
01	00	Normal memory	Non Cacheable (NC)
01	!=00	Normal memory	Write-Back (WB)
10	xx	Normal memory	Write-Through (WT)
11	xx	Normal memory	Write-Back (WB)

For more information, see chapter *The AArch64 Virtual Memory System Architecture* of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

C1.7.2   Access control

The access permission and security configuration bits, such as AP, XN, and NS in the VMSAv8-64 translation table descriptors are present in the MPU configuration registers in PMSAv8-64.

Access granted in the stage 1 of the Secure EL1&0 translation regime by the EL1 MPU is subject to further qualification by the EL2 MPU in the stage 2 of the Secure EL1&0 translation regime.

The AP, XN, and NS bits in PMSAv8-64 are interpreted in the same way as defined by VMSAv8-64. For selecting memory attributes and defining access permissions, PMSAv8-64 follows the same prioritization rules defined by VMSAv8-64 in Armv8-A.

PMSAv8-64 does not support hierarchical control bits defined in the VMSAv8-64 table descriptors. PMSAv8-64 also does not support Privileged execute-never (PXN) and Unprivileged execute-never (UXN) bits. PMSAv8-64 behaves as PXN = XN and UXN = XN, and follows the same rule defined by VMSAv8-64 in combining permission attributes.

If the value of SCTLR_ELx.{M, BR} is {0, 1}, where ELx is the highest Exception level of the translation regime, then PE behaves as if the value of SCTLR_ELx.WXN is 0 for all purposes other than returning the value of a direct read of the field.

If the value of SCTLR_EL1.{M, BR} is {0, 1}, then for the Secure EL1&0 translation regime, any memory region that is writable at EL0, is also executable from EL1 if that address is marked as executable by the Background region.

Armv8-R AArch64 supports FEAT_PAN as defined by the Armv8-A architecture. If the value of SCTLR_EL1.M is 1, FEAT_PAN is applied to the stage 1 of the Secure EL1&0 translation regime.

C1.7.3 Memory attributes and access permission mappings

Memory attributes, Security states, and access permission information present in the translation table descriptors of VMSAv8-64 are mapped to MPU configuration registers in PMSAv8-64, as described in Table C1-13.

Table C1-13 Memory attributes and access permission mappings

Attributes and permissions	MPU register fields	Description
Non-secure (NS)	PRLAR_EL1.NS PRLAR_EL2.NS	Specifies whether the translated address is in the Secure or Non-secure address space.
Access Permission (AP)	PRBAR_EL1.AP PRBAR_EL2.AP	Defines the Access permissions for the protection region.
Execute Never (XN)	PRBAR_EL1.XN PRBAR_EL2.XN	Defines the Execute-never attribute for the protection region.
Shareability (SH)	PRBAR_EL1.SH PRBAR_EL2.SH	Defines the Shareability for a Normal memory region. For any type of Device memory or Normal Non-cacheable memory, the value of the SH[1:0] field is IGNORED.
Attribute Index (AttrIndx)	PRLAR_EL1.AttrIndx PRLAR_EL2.AttrIndx	Indexes an Attr<n> field in the MAIR_EL1 or MAIR_EL2 register, which gives the memory type and memory attributes.

C1.7.4 Combining memory attributes and access permissions

Armv8-R AArch64 uses the same architecture rules as Armv8-A for combining stage 1 and stage 2 memory attributes and the access permissions for the Secure EL1&0 translation regime.

Stage 2 forced Write-Back Feature

If FEAT_S2FWB is enabled (HCR_EL2.FWB=1), the Inner and Outer Memory attributes for the stage 2 of the Secure EL1&0 translation regime must be the same with the same encoding, otherwise the combined attribute is UNKNOWN.

- If FEAT_S2FWB is enabled and the memory attributes for the stage 2 of the Secure EL1&0 translation regime are Write-Back (WB) with Attr[7:6] = 0b11, then the combined attribute is WB. If the memory attribute is assigned by the MPU, the attribute encoding is derived from MAIR_EL2.Attr[7:6]. If the memory attribute is assigned from Background region, then the encoding is derived from the memory region configuration in the Background region.

- If FEAT_S2FWB is not enabled, then stage 1 and stage 2 memory attributes of the Secure EL1&0 translation regime are combined using Armv8-A rule for combining memory attributes when FEAT_S2FWB is not enabled.

——— **Note** ———

In Armv8-A, the memory attribute encoding that enables FEAT_S2FWB is MemAttr[4:2] = 0b110 in stage 2 block or page descriptor of the Secure EL1&0 translation regime, while in Armv8-R AArch64, it is MAIR_EL2.Attr[7:6] = 0b11. Therefore, FEAT_S2FWB architecture rules defined for the 0b110 encoding in Armv8-A must be applied to the 0b11 encoding in Armv8-R AArch64.

For more information, see the *Stage 2 memory region type and Cacheability attributes when Armv8.4-S2FWB is implemented* section of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

C1.7.5 Enabling and disabling the caching of memory accesses

The Armv8-R AArch64 architecture follows the same rule as the Armv8-A AArch64 architecture for enabling and disabling cache, and can force all memory locations with the Normal memory type to be treated as Non-cacheable regardless of their assigned Cacheability attribute using SCTLR_ELx.{I,C} and HCR_EL2.{ID,CD}. For a PMSAv8-64 based address translation, Armv8-R AArch64 extends the functionality of SCTLR_ELx.{I,C} and HCR_EL2.{ID,CD} for accesses to Background region also.

C1.7.6 Enabling and disabling stages in translation regimes

The EL1 MPU controls the stage 1 of the Secure EL1&0 translation regime. The EL2 MPU controls the stage 2 of the Secure EL1&0 translation regime and the stage 1 of the Secure EL2 translation regime.

For the Secure EL1&0 translation regime:
- If SCTLR_EL1.{M, BR} is {0, 0}, then stage 1 translation is disabled.
- If HCR_EL2.VM is 0, then stage 2 translation is disabled.
- If HCR_EL2.VM is 1 and SCTLR_EL2.{M, BR} is {0, 0}, then the behavior is CONSTRAINED UNPREDICTABLE with the following permitted values:
  — The stage 2 memory attribute becomes UNKNOWN.
  — Raise stage 2 level 0 translation fault.

For the Secure EL2 translation regime if SCTLR_EL2.{M, BR} is {0, 0}, then translation is disabled.

C1.8 MPU fault encodings

PMSAv8-64 defines MPU faults that are consistent with the VMSAv8-64 fault definitions and reuses IFSC and DFSC fault encodings. Each MPU raises faults on invalid memory accesses, including accesses to memory regions outside address ranges and the accesses without sufficient permissions. Table C1-14 describes the MPU fault encodings mapped from VMSAv8-64.

Table C1-14 PMSAv8-64 fault encodings

Memory faults	MPU fault encodings mapped from VMSAv8-64	Description
Alignment fault	Alignment fault	Unaligned memory access.
Translation fault	Translation fault, level 0	Invalid input address. There is no valid mapping or valid memory attributes for the input address.
Permission fault	Permission fault, level 0 (0b001100)	Insufficient access permissions.
Address size fault	Address size fault, level 0	Generated output address is out of range of the implemented physical address.
Access flag fault		Not applicable for PMSAv8-64.
TLB conflict abort		Not applicable for PMSAv8-64.
Synchronous abort (translation table walk)		Not applicable for PMSAv8-64.

C1.8.1 See also

In the Arm Architecture Reference Manual

- Program counter and stack pointer alignment.
- FEAT_LSE, Large System Extensions.

C1.9 PMSAv8-64 implications for caches

Enabling, reconfiguring, or reprogramming any memory region can result in new and different memory attributes for a previously accessed or speculatively accessed address. In this case, the rules for mismatched memory attributes apply. See the *Mismatched memory attributes* section of the *Arm® Architecture Reference Manual Armv8, for the Armv8-A architecture profile*.

C1.9.1 Cache line length

The PMSAv8-64 permits the definition of memory regions that might be smaller than a cache line in the implementation. Therefore, the following rules apply:

- If the MPU is configured so that multiple differing attributes apply to a single cache line, then for any access to that cache line, the rules for mismatched memory attributes apply.
- Marking any part of a cache line as Write-Back permits the entire line to be treated as Write-Back.

C1.10 Address tagging and pointer authentication support

The Armv8-R AArch64 architecture supports Address tagging defined in Armv8-A architecture. In PMSAv8-64, TCR_EL1.TBI1 is RAZ.

The Armv8-R AArch64 architecture supports FEAT_PAuth2 feature defined in Armv8-A architecture with a modified definition of PAC field as described below:

* When Address tagging is used, the PAC field is Xn[54:bottom_PAC_bit].
* When Address tagging is not used, the PAC field is Xn[63:56, 54:bottom_PAC_bit].

In PMSAv8-64, the bottom_PAC_bit is the maximum physical address size as indicated by ID_AA64MMFR0_EL1.PARange.

C1.10.1 See also

In the Arm Architecture Reference Manual

* Features added to the Armv8.3 extension in later releases.
* Pointer authentication in AArch64 state.
* System register control of pointer authentication.

C1.11   Security model

The Armv8-R AArch64 architecture defines the security model based on Armv8-A AArch64 with the following changes:

- The Armv8-R AArch64 architecture does not support EL3.
- The Armv8-R AArch64 architecture always executes code in Secure state with EL2 as the highest Exception level and:
    — It is not possible to switch to Non-secure state.
    — EL0, EL1, and EL2, all run in Secure state.

C1.11.1   Secure EL2

The Armv8-R AArch64 architecture adopts FEAT_SEL2 feature from Armv8.4-A architecture extension and introduces it as PMSAv8-64 based architecture feature. The protection region configuration registers for EL1 and EL2 MPUs, PRLAR_EL1 and PRLAR_EL2, are extended to include the NS bit that has the same behavior as NS bit in the translation table descriptor of VMSAv8-64. Security controls for translation table walks are not supported in PMSAv8-64.

The NS bit in the PRLAR_EL1 and PRLAR_EL2 specifies whether the output address is in the Secure or Non-secure address space. Each protection region can be independently configured to the Secure or Non-secure address space.

The configuration bits required to implement Secure and Non-secure states in VMSAv8-64 are mapped to PMSAv8-64 as described in Table C1-15.

Table C1-15 VMSAv8-64 and PMSAv8-64 security configuration

VMSAv8-64	PMSAv8-64
NS bit in translation table	PRLAR_EL1.NS/PRLAR_EL2.NS
NSTable bit in translation table	NA
VSTCR_EL2.SW	NA
VSTCR_EL2.SA	VSTCR_EL2.SA
VTCR_EL2.NSW	NA
VTCR_EL2.NSA	VTCR_EL2.NSA

See also

In the Arm Architecture Reference Manual

- The VMSAv8-64 address translation system.
- VMSAv8-64 translation table format descriptors.
- Security state of translation table lookups.
- Translation tables and the translation process.

Secure EL1&0 translation

As the stage 2 of the Secure EL1&0 translation regime is controlled by the EL2 MPU and there is no page-based address translation or translation table walk, the following modifications are added to Armv8-R AArch64 FEAT_SEL2:

- VTCR_EL2.NSW and VSTCR_EL2.SW control bits are RES0.
- VSTTBR_EL2 register is not present.
- Adds a secure check control, VSTCR_EL2.SC.

In addition, VSTCR_EL2.SA and VTCR_EL2.NSA controls are also supported and have the same functionality as FEAT_SEL2 but are applied to PA in the Secure EL1&0 translation regime.

VSTCR_EL2.SC has no effect on the Secure EL2 translation regime. Table C1-16 describes the behavior of VSTCR_EL2.SC for the Secure EL1&0 translation regime.

Table C1-16 Secure check behavior in the Secure EL1&0 translation regime

VA	EL1 MPU NS attribute	IPA	EL2 MPU NS attribute	SC	PA
Secure	0	Secure	0	x	Secure
Secure	1	Non-secure	1	x	Non-secure
Secure	0	Secure	1	0	Non-secure
Secure	0	Secure	1	1	Fault
Secure	1	Non-secure	0	0	Non-secure
Secure	1	Non-secure	0	1	Fault

EL1 and EL0 access is further subjected to VSTCR_EL2.SA and VTCR_EL2.NSA controls as defined by Armv8.4-A.

Table C1-17 describes the behavior of VSTCR_EL2.SA and VTCR_EL2.NSA in the Secure EL1&0 translation regime.

Table C1-17 VSTCR_EL2.SA and VTCR_EL2.NSA behavior in the Secure EL1&0 translation regime

PA.NS	VSTCR_EL2.SA	VTCR_EL2.NSA	PA.NS (final)
0	0	x	0
0	1	Behaves as 1	1
1	0	0	0
1	0	1	1
1	1	x	1

The Armv8-R AArch64 architecture follows the Armv8-A architecture rules on whether VSTCR_EL2.SA and VTCR_EL2.NSA controls should be applied.
- If HCR_EL2.VM=1 and SCTLR_EL2.M=1 or SCTLR_EL2.BR=1, then stage 2 Secure EL1&0 translation is enabled.
- If HCR_EL2.VM=0, then stage 2 Secure EL1&0 translation is disabled.

C1.11.2 Secure EL2 translation

The Armv8-R AArch64 architecture implements PMSAv8-64 at the stage 1 of the Secure EL2 translation regime. Therefore, there is no translation table walk and address mapping. The NS bit in the EL2 MPU protection region register determines whether the output address must be looked in Secure or Non-secure address space.

C1.11.3 See also

In the Arm Architecture Reference Manual
- The Armv8-A security model.

- The AArch64 System Level Programmers' Model.
- The Armv8.4 architecture extension.
- ARMv8.4-SecEL2, Secure EL2.

C1.12   Virtualization

Armv8-R AArch64 implements a permission-based containerization by introducing a stage 2 translation using an MPU controlled by a hypervisor running at EL2. The MPU does not perform address mapping and only checks permissions. Therefore, Armv8-R AArch64 relies on hypervisor configured permission attributes of the memory region to implement containerization.

For more information, see chapter *The AArch64 System Level Programmers' Model* of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

C1.12.1   Support for Guest operating systems

The hypervisor supports Guest operating systems using PMSAv8-64 on a per guest basis. PMSAv8-64 guests have access to the EL1 MPU for sandboxing individual tasks. Memory accesses by PMSAv8-64 guests are further validated by the EL2 MPU, controlled by the hypervisor.

If multiple PMSAv8-64 guests are present, then these guests must be configured to use non-conflicting physical memory addresses. Virtualization is supported by the EL2 MPU at the stage 2 of the Secure EL1&0 translation regime.

Part D

Armv8-R AArch64 Virtual Memory System Architecture

Chapter D1
Virtual Memory System Architecture

This chapter provides a system-level view of the Virtual Memory System Architecture (VMSAv8-64) for any implementation that is compliant with the Armv8-R AArch64 architecture. It contains the following sections:

- *About the Virtual Memory System Architecture* on page D1-64.
- *Architecture extensions in VMSAv8-64* on page D1-65.
- *Support for VMSAv8-64 in Armv8-R AArch64* on page D1-66.
- *System registers access control* on page D1-67.
- *Virtualization* on page D1-68.
- *System operations* on page D1-69.

———— **Note** ————

The information related to VMSAv8-64 as described in this chapter is at Beta quality.

D1.1 About the Virtual Memory System Architecture

This chapter describes the Virtual Memory System Architecture (VMSAv8-64) for the Armv8-R AArch64 architecture profile.

VMSAv8-64 provides a Memory Management Unit (MMU) that controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the PE. The process of address translation maps the virtual addresses (VAs) used by the PE onto the physical addresses (PAs) of the physical memory system.

Armv8-R AArch64 supports VMSAv8-64 as an optional memory system architecture at the stage 1 of the Secure EL1&0 translation regime, and supports general purpose operating systems, such as Linux and Android at EL1. With VMSAv8-64 supported at EL1, the Armv8-R AArch64 architecture profile can have the following memory system configurations:

- PMSAv8-64 at EL1 and EL2.
- PMSAv8-64 or VMSAv8-64 at EL1, and PMSAv8-64 at EL2.

For more information, see *The AArch64 Virtual Memory System Architecture* chapter of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

D1.2   Architecture extensions in VMSAv8-64

Implementations of Armv8-R AArch64 with VMSAv8-64 at EL1 support the following Armv8-A features listed in Table D1-1.

Table D1-1 Armv8-A features supported for implementations with VMSAv8-64

Feature	Description
FEAT_HPDS	Hierarchical permission disables
FEAT_HAFDBS	Hardware management of the Access flag and dirty state
FEAT_HPDS2	Translation table page-based hardware attributes
FEAT_TTCNP	Translation table Common not private translations
FEAT_TTL	Translation Table Level
FEAT_BBM	Translation table break-before-make levels
FEAT_TTST	Small translation tables
FEAT_E0PD	Preventing EL0 access to halves of address maps
FEAT_TLBIOS	TLB invalidate instructions in Outer Shareable domain
FEAT_TLBIRANGE	TLB invalidate range instructions

Control fields for the architecture features, which are RES0 in some contexts if the value of VTCR_EL2.MSA = 0, are treated as RES0 in all contexts if an implementation does not support VMSAv8-64. For the architectural features supported by Armv8-R AArch64, whether a feature is mandatory or optional depends on whether the feature is mandatory or optional in the Armv8.4-A architecture.

D1.3   Support for VMSAv8-64 in Armv8-R AArch64

The MSA and MSA_frac fields in the ID_AA64MMFR0_EL1 register identify the memory system configurations supported at EL1.

In Armv8-R AArch64, the only permitted value for ID_AA64MMFR0_EL1.MSA is 0b1111. When ID_AA64MMFR0_EL1.MSA_frac is 0b0010, the stage 1 of the Secure EL1&0 translation regime can enable PMSAv8-64 or VMSAv8-64 architecture.

If PE supports both PMSAv8-64 and VMSAv8-64 at EL1, then VTCR_EL2.MSA determines the memory system architecture enabled at the stage 1 of the Secure EL1&0 translation regime. Depending on the memory system architecture, the stage 1 of the Secure EL1&0 translation regime is controlled by either an EL1 MPU for PMSAv8-64, or an MMU for VMSAv8-64.

The stage 2 of the Secure EL1&0 translation regime and the stage 1 of the Secure EL2 translation regime are controlled by EL2 MPU. Armv8-R AArch64 uses the same translation table format and fault encodings as Armv8-A.

For more information, see *The AArch64 Virtual Memory System Architecture* chapter of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

D1.4   System registers access control

When EL1 is running PMSAv8-64 based Guest operating system, access to TTBR1_EL1 is UNDEFINED. When EL1 is running a VMSAv8-64 based Guest operating system, EL1 access to the PMSAv8-64 registers is UNDEFINED.

The following EL1 PMSAv8-64 registers are UNDEFINED from EL1 in a VMSA context:

*   MPUIR_EL1.
*   PRBAR_EL1.
*   PRBAR<n>_EL1.
*   PRLAR_EL1.
*   PRLAR<n>_EL1.
*   PRSELR_EL1.
*   PRENR_EL1.

─── **Note** ───

TTBR1_EL1 is UNDEFINED from EL1 in a PMSA context. If an implementation supports only PMSAv8-64 at EL1, then accessing VMSAv8-64 register, TTBR1_EL1, is UNDEFINED from both EL1 and EL2.

─────

Both VMSAv8-64 and PMSAv8-64 registers are accessible from EL2 independent of whether a Guest operating system at EL1 uses PMSAv8-64 or VMSAv8-64.

D1.5 Virtualization

For Armv8-R AArch64, the hypervisor running at EL2 selects the memory system architecture for each Guest operating system by configuring the VTCR_EL2.MSA bit. This enables the hypervisor to support multiple Guest operating systems utilizing either PMSAv8-64 or VMSAv8-64 on a per guest basis. Memory accesses by both VMSAv8-64 and PMSAv8-64 guests at EL1 are further validated at the stage 2 of the Secure EL1&0 translation regime.

If multiple VMSAv8-64 or PMSAv8-64 guests are present, then these guests must be configured to use non-conflicting physical memory addresses.

—— **Note** ——

Secure Check control, VSTCR_EL2.SC does not differentiate between translation table walk or memory access.

For more information, see *The AArch64 Virtual Memory System Architecture* chapter of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

D1.6   System operations

D1.6.1   Address translation instructions

When executed from EL2, address translation instructions use VTCR_EL2.MSA to determine whether the EL1 context is PMSAv8-64 or VMSAv8-64.

D1.6.2   TLB maintenance instructions

Armv8-R AArch64 permits an implementation to cache stage 1 VMSAv8-64 and stage 2 PMSAv8-64 attributes as a common entry for the Secure EL1&0 translation regime.

Stage 1 VMSAv8-64 is permitted to cache stage 2 PMSAv8-64 MPU configuration as a part of the translation process. Visibility of stage 2 MPU updates for stage 1 VMSAv8-64 contexts requires associated TLB invalidation for stage 2. The stage 2 TLB invalidation is not required to apply to caching structures that combine stage 1 and stage 2 attributes.

D1.6.3   See also

In the Arm Architecture Reference Manual

• A64 Instruction Set Overview.
• The A64 System Instruction Class.
• A64 System instructions for TLB maintenance.

Part E

A64 Instruction Set for Armv8-R AArch64

Chapter E1
A64 Instruction Set for Armv8-R AArch64

This chapter describes the instructions in Armv8-R AArch64. It contains the following sections:

- *Instruction encodings* on page E1-74.
- *A64 instructions in Armv8-R AArch64* on page E1-75.

E1.1 Instruction encodings

This section contains the encodings for the Armv8-R AArch64 instructions. The encodings in this section are decoded from the Branches, Exception Generating, and System instructions chapter of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 8	7 5	4 0
1 1 0 1	0 1 0 1	0 0 0 0	0 0 1 1	0 0 1 1	CRm	op2	Rt
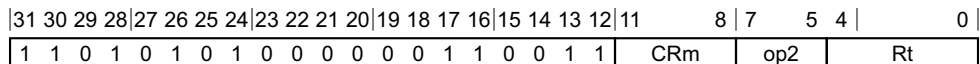
Table E1-1 Decode fields and Instructions

Decode fields			Instruction Details
CRm	**op2**	**Rt**	
-	000	-	Unallocated
-	001	!= 11111	Unallocated
-	010	11111	CLREX
-	101	11111	DMB
-	110	11111	ISB
-	111	!= 11111	Unallocated
-	111	11111	SB
!= 0x00	100	11111	DSB
0000	100	11111	SSBB
0001	011	-	Unallocated
001x	011	-	Unallocated
01xx	011	-	Unallocated
0100	100	11111	PSSBB
1xxx	011	-	Unallocated

E1.2 A64 instructions in Armv8-R AArch64

This section describes the A64 instructions in Armv8-R AArch64.

For more information, see Definition of the Armv8 memory model in the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

E1.2.1 Data Full Barrier

Data Full Barrier is a memory barrier that ensures the completion of memory accesses. If executed at EL2, this instruction orders memory accesses irrespective of their Exception level or associated VMID. If executed at EL1 or EL0, this instruction behaves as DSB SY.

This instruction is an alias of the DSB instruction. This means that:

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

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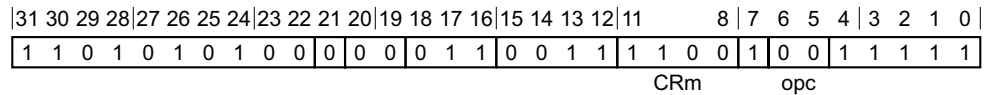
Encoding

DFB

is equivalent to

DSB #12

and is always the preferred disassembly.

Operation

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

E1.2.2 Data Memory Barrier

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses.

The ordering requirements for Data Memory Barrier are as follows:

- EL1 and EL0 memory accesses are ordered only with respect to the memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.

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

opc

Encoding

```
DMB <option>|#<imm>
```

Decode for this encoding

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

Assembler symbols

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

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

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

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

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

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

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

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

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

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

OSH	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.
OSHST	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.
OSHLD	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of `CRm` that are not listed above are reserved, and can be encoded using the `#<imm>` syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* in *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile* or see *Data Synchronization Barrier (DSB)* in *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

`<imm>`	Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
vmid_sensitive = (PSTATE.EL != EL2) && (CRm<1:0> != '00');
DataMemoryBarrier(domain, types, vmid_sensitive);
```

E1.2.3 Data Synchronization Barrier

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses.

The ordering requirements for Data Synchronization Barrier are as follows:

- EL1 and EL0 memory accesses are ordered only with respect to the memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.

This instruction is used by the alias DFB. See *Alias conditions* for details of when each alias is preferred.

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

Encoding

```
DSB <option>|#<imm>
```

Decode for this encoding

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

Alias conditions

Alias	is preferred when
DFB	CRm == '1100' && op2 == '100'

Assembler symbols

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

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

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

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

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

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

ISHLD	Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.
NSH	Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.
NSHST	Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.
NSHLD	Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.
OSH	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.
OSHST	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.
OSHLD	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* in *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile* or see *Data Synchronization Barrier (DSB)* in *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

—— **Note** ——

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

<imm>	Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
vmid_sensitive = (PSTATE.EL != EL2) && (CRm<1:0> != '00');
DataSynchronizationBarrier(domain, types, vmid_sensitive);
```

Part F

The A64 System Instructions

Chapter F1
The A64 System Instructions

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following section:

F1.1 System instructions

The Armv8-R AArch64 architecture supports all System instructions defined by Armv8-A. The behavior of these instructions is the same as that in Armv8-A with no EL3. The MPU register updates are guaranteed to be visible to all PMSAv8-64 translation regimes following a Context synchronization event operation.

F1.1.1 Address translation instructions

In PMSAv8-64, the VA, IPA, and PA are all the same and translation operations are reduced to memory attributes, security checks, and permission checks. If multiple memory regions are enabled in a 4KB memory boundary, AT instruction gives correct memory region attributes in a successful transaction and PAR_EL1 provides only 4KB memory boundary aligned address.

For more information, see *A64 Instruction Set Overview* and *The A64 System Instruction Class* chapters of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

Part G

Armv8-R AArch64 System Registers

Chapter G1
System Registers in a PMSA Implementation

This chapter describes the system control registers in a PMSA implementation. The registers are described in alphabetical order. It contains the following sections:

G1.1 System register groups

System registers provide control and status information of architected features. Armv8-R AArch64 System registers are grouped according to whether they only exist in the Armv8-R AArch64 profile or whether they have been modified from the equivalent Armv8-A System registers. All other registers are described in *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

Table G1-1 lists the Armv8-R AArch64 registers.

Table G1-1 Alphabetical index of Armv8-R AArch64 registers

Register	Description
MPUIR_EL1	MPU Type Register (EL1)
MPUIR_EL2	MPU Type Register (EL2)
PRBAR_EL1	Protection Region Base Address Register (EL1)
PRBAR_EL2	Protection Region Base Address Register (EL2)
PRBAR<n>_EL1	Protection Region Base Address Registern (EL1)
PRBAR<n>_EL2	Protection Region Base Address Registern (EL2)
PRENR_EL1	Protection Region Enable Register (EL1)
PRENR_EL2	Protection Region Enable Register (EL2)
PRLAR_EL1	Protection Region Limit Address Register (EL1)
PRLAR_EL2	Protection Region Limit Address Register (EL2)
PRLAR<n>_EL1	Protection Region Limit Address Registern (EL1)
PRLAR<n>_EL2	Protection Region Limit Address Registern (EL2)
PRSELR_EL1	Protection Region Selection Register (EL1)
PRSELR_EL2	Protection Region Selection Register (EL2)
VSCTLR_EL2	Virtualization System Control Register (EL2)

Table G1-2 lists the modified AArch64 registers.

Table G1-2 Alphabetical index of modified AArch64 System registers

Register	Description
CPACR_EL1	Architectural Feature Access Control Register
CPTR_EL2	Architectural Feature Trap Register (EL2)
DBGBCR<n>_EL1	Debug Breakpoint Control Registers
HCR_EL2	Hypervisor Configuration Register
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0

Table G1-2 Alphabetical index of modified AArch64 System registers (continued)

Register	Description
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDSCR_EL1	Monitor Debug System Control Register
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCR_EL0	Performance Monitors Control Register
PMEVTYPER<n>_EL0	Performance Monitors Event Type Registers
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
VTCR_EL2	Virtualization Translation Control Register
VSTCR_EL2	Virtualization Secure Translation Control Register

Table G1-3 lists the external registers.

Table G1-3 Alphabetical index of modified external registers

Register	Description
DBGBCR<n>_EL1	Debug Breakpoint Control Registers
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDPFR	External Debug Processor Feature Register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCR_EL0	Performance Monitors Control Register
PMEVTYPER<n>_EL0	Performance Monitors Event Type Registers

Table G1-4 lists the Armv8-A System registers that are not supported in Armv8-R AArch64.

Table G1-4 Alphabetical index of System registers that are not supported in Armv8-R AArch64

Register	Description
VSTTBR_EL2	Virtualization Secure Translation Table Base Register
VTTBR_EL2	Virtualization Translation Table Base Register
TTBR0_EL2	Translation Table Base Register 0 (EL2)
TTBR1_EL2	Translation Table Base Register 1 (EL2)
TTBR1_EL1	Translation Table Base Register 1 (EL1)[a]

> a.   TTBR1_EL1 is present only in the VMSAv8-64 extension.

——— **Note** ———

The exceptions caused by an EL1 register access being UNDEFINED is at priority level 18. Hence, any exception taken to EL2 because of HCR_EL2.{TID1, TVM, TRVM} trap controls, has higher priority, priority level 16. The VSCTLR_EL2 register reuses the encoding of TTBR0_EL2 in Armv8-A.

Table G1-5 disambiguates the general names of the PMSA memory region registers used in this chapter.

Table G1-5 Disambiguation of PMSA memory region registers by Exception level

General form	EL0	EL1	EL2
PRBAR<n>_ELx	-	PRBAR<n>_EL1	PRBAR<n>_EL2
PRBAR_ELx	-	PRBAR_EL1	PRBAR_EL2
PRLAR<n>_ELx	-	PRLAR<n>_EL1	PRLAR<n>_EL2
PRLAR_ELx	-	PRLAR_EL1	PRLAR_EL2
PRSELR_ELx	-	PRSELR_EL1	PRSELR_EL2
MPUIR_ELx	-	MPUIR_EL1	MPUIR_EL2
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2

G1.1.1   See also

In the Arm Architecture Reference Manual

- AArch64 System Register Descriptions.

G1.2   Accessing MPU memory region registers

The MPU memory region in PMSAv8-64 is defined by a set of registers, PRBAR_ELx and PRLAR_ELx. These registers define the memory region base address, memory region size, and memory attributes. Each MPU region can be independently configured and the MPUIR_ELx register identifies the actual number of implemented regions.

The MPU provides two register interfaces to program the MPU regions:

* Access to any of the MPU regions via PRSELR_ELx, PRBAR<n>_ELx, and PRLAR<n>_ELx.
* Access to MPU regions at offsets from the aligned value of PRSELR_ELx.REGION via PRBAR_ELx and PRLAR_ELx.

When n=0, the encoding of PRBAR<n>_ELx and PRLAR<n>_ELx corresponds to PRBAR_ELx and PRLAR_ELx respectively.

When n != 0, then the encoding of PRBAR<n>_ELx and PRLAR<n>_ELx corresponds to the configuration of m-th MPU region:

```
m = r: n

where r = PRSELR_ELx.REGION<7:4> and
n IN {0..15}
```

Access to MPU region registers beyond the number of implemented regions is CONSTRAINED UNPREDICTABLE. The value of n can be between 0 and 15 and is encoded using the CRm and op2 fields.

Table G1-6 Register encoding scheme for PRBAR<n>_ELx and PRLAR<n>_ELx

Register	CRm	op2
PRBAR<n>_ELx	1:n<3:1>	n<0>:00
PRLAR<n>_ELx	1:n<3:1>	n<0>:01

G1.3   General system control registers

This section lists the System registers in Armv8-R AArch64 that are not part of one of the other listed groups.

G1.3.1 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, SVE, Advanced SIMD and floating-point functionality.

Configurations

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.

Attributes

CPACR_EL1 is a 64-bit register.

Field descriptions

The CPACR_EL1 bit assignments are:
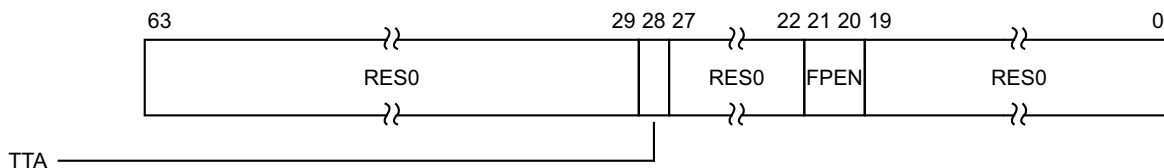
Bits [63:29]

Reserved, RES0.

TTA, bit [28]

Traps EL0 and EL1 System register accesses to all implemented trace registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.{E2H, TGE} is {0,1}, from both Execution states as follows:

- In AArch64 state, accesses to trace registers are trapped, reported using EC syndrome value 0x18.

- In AArch32 state, MRC and MCR accesses to trace registers are trapped, reported using EC syndrome value 0x05.

- In AArch32 state, MRRC and MCRR accesses to trace registers are trapped, reported using EC syndrome value 0x0C.

0b0   This control does not cause any instructions to be trapped.

0b1   This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.

> **———— Note ————**
>
> - The ETMv4 architecture does not permit EL0 to access the trace registers. If the PE trace unit implements FEAT_ETMv4, 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.

This field resets to an architecturally UNKNOWN value.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps EL0 and EL1 accesses to the SVE, Advanced SIMD, and floating-point registers to EL1, reported using EC syndrome value 0x07, or to EL2 reported using EC syndrome value 0x00, when EL2 is implemented and enabled for the current Security state and HCR_EL2.{E2H, TGE} is {0,1}, from both Execution states as follows:

- In AArch64 state, accesses to FPCR, FPSR, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.

- In AArch32 state, FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers.

0b00   This control causes any instructions at EL0 or EL1 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN.

0b01   This control causes any instructions at EL0 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN, but does not cause any instruction at EL1 to be trapped.

0b10   This control causes any instructions at EL0 or EL1 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN.

0b11   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 0b11.

————————————

This field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Accessing the CPACR_EL1

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.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPTR_EL2.TCPAC == '1' then
```

```
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                return CPACR_EL1;
        elsif PSTATE.EL == EL2 then
            return CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    CPACR_EL1 = X[t];
```

G1.3.2   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 SVE, Advanced SIMD and floating-point functionality.

- EL2 access to trace functionality, and to SVE, Advanced SIMD and floating-point functionality.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

CPTR_EL2 is a 64-bit register.

Field descriptions

The CPTR_EL2 bit assignments are:
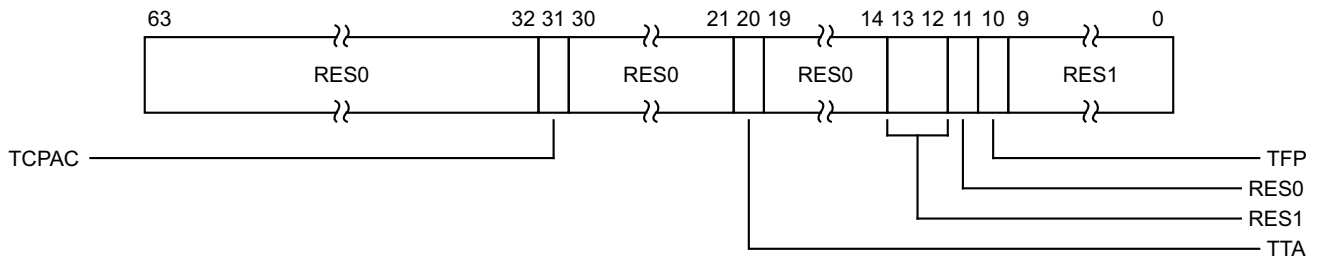
This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Traps EL1 accesses to CPACR_EL1, reported using EC syndrome value `0x18` and accesses to CPACR, reported using EC syndrome value `0x03`, to EL2 when EL2 is enabled in the current Security state.

`0b0`   This control does not cause any instructions to be trapped.

`0b1`   EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2 when EL2 is enabled in the current Security state.

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.

—————————————

This field resets to an architecturally UNKNOWN value.

Bits [30:21]

Reserved, RES0.

TTA, bit [20]

Traps System register accesses to all implemented trace registers to EL2 when EL2 is enabled in the current Security state, from both Execution states as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1 are trapped to EL2, reported using EC syndrome value `0x18`.

- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1 are trapped to EL2, reported using EC syndrome value `0x05`.

- In AArch32 state, MRRC or MCRR accesses to trace registers with cpnum=14, opc1=1 are trapped to EL2, reported using EC syndrome value `0x0C`.

`0b0`   This control does not cause any instructions to be trapped.

`0b1`   Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by CPACR.TRCDIS or CPACR_EL1.TTA.

─── **Note** ───

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the PE trace unit implements FEAT_ETMv4, 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.

This field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps accesses to SVE, Advanced SIMD and floating-point functionality to EL2 when EL2 is enabled in the current Security state, from both Execution states, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x07`:

  — FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.

- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x07`:

  — MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

`0b0`   This control does not cause any instructions to be trapped.

0b1   Any attempt at EL0, EL1 or EL2, to execute an instruction that uses the registers
             associated with SVE, Advanced SIMD and floating-point execution is trapped to EL2
             when EL2 is enabled in the current Security state, subject to the exception prioritization
             rules, unless it is trapped by CPTR_EL2.TZ.

————— **Note** —————

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

————————————————

This field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES1.

Accessing the CPTR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, CPTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return CPTR_EL2;
```

MSR CPTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CPTR_EL2 = X[t];
```

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```

```
        return CPACR_EL1;
elsif PSTATE.EL == EL2 then
    return CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    CPACR_EL1 = X[t];
```

G1.3.3 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

Purpose

> Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configurations

> If EL2 is not implemented, this register is RES0 from EL3.
>
> The bits in this register behave as if they are 0 for all purposes other than direct reads of the register if EL2 is not enabled in the current Security state.

Attributes

> HCR_EL2 is a 64-bit register.

Field descriptions

The HCR_EL2 bit assignments are:
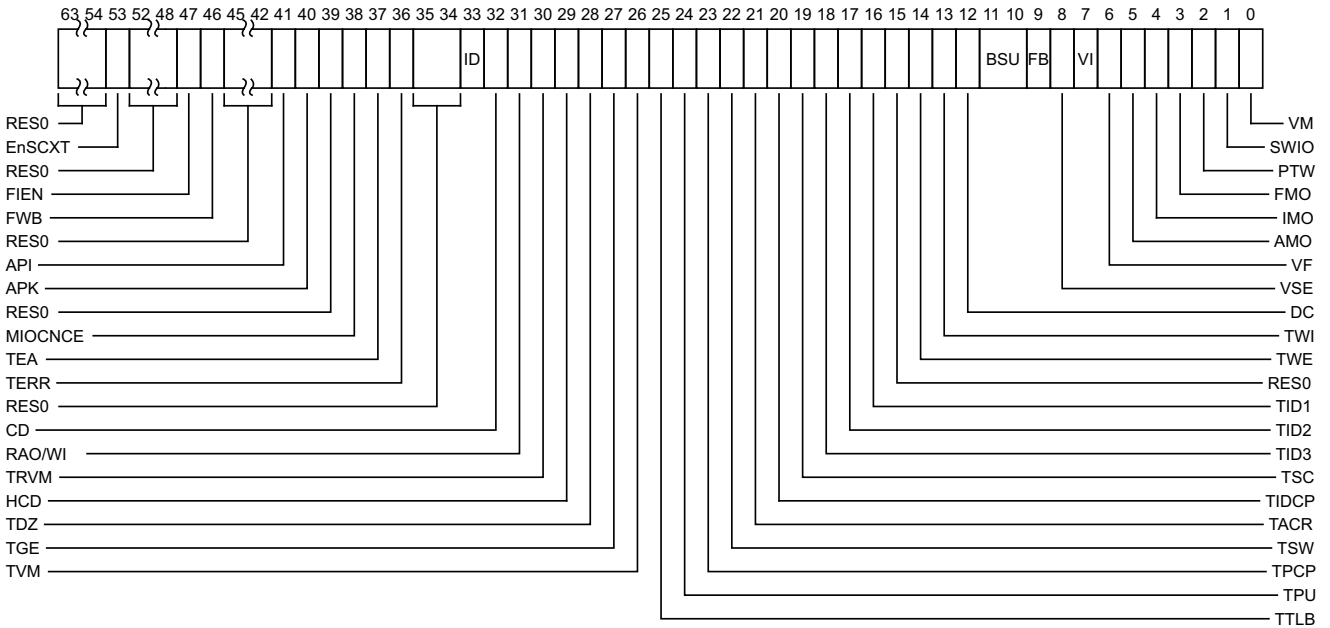
Bits [63:54]

> Reserved, RES0.

EnSCXT, bit [53]

> *When FEAT_CSV2 is implemented:*
>
> Enable Access to the SCTXNUM_EL1 and SCTXNUM_EL0 registers. The defined values are:
>
> 0b0   When (HCR_EL2.TGE==0 or HCR_EL2.E2H==0) and EL2 is enabled in the current Security state, EL1 and EL0 access to SCTXNUM_EL0 and EL1 access to SCTXNUM_EL1 is disabled by this mechanism, causing an exception to EL2, and the values of these registers to be treated as 0.
>
>   When ((HCR_EL2.TGE==1 and HCR_EL2.E2H==1) and EL2 is enabled in the current Security state, EL0 access to SCTXNUM_EL0 is disabled by this mechanism, causing an exception to EL2, and the value of this register to be treated as 0.

0b1   This control does not cause accesses to SCTXNUM_EL0 or SCTXNUM_EL1 to be
            trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has
no effect on execution at EL0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [52:48]

Reserved, RES0.

FIEN, bit [47]

When FEAT_RASv1p1 is implemented:

Fault Injection Enable. Unless this bit is set to 1, accesses to the ERXPFGCDN_EL1,
ERXPFGCTL_EL1, and ERXPFGF_EL1 registers from EL1 generate a Trap exception to EL2,
when EL2 is enabled in the current Security state, reported using EC syndrome value 0x18.

0b0   Accesses to the specified registers from EL1 are trapped to EL2, when EL2 is enabled
            in the current Security state.

0b1   This control does not cause any instructions to be trapped.

If EL2 is disabled in the current Security state, the Effective value of HCR_EL2.FIEN is 0b1.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record
accessible using System registers is owned by a node that implements the RAS Common Fault
Injection Model Extension, then this bit might be RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FWB, bit [46]

When FEAT_S2FWB is implemented:

Forced Write-Back. Defines the combined cacheability attributes in a 2 stage translation regime.

0b0   When this bit is 0, then:

   •   The combination of stage 1 and stage 2 translations on memory type and
       cacheability attributes are as described in the Armv8.0 architecture. For more
       information see 'Combining the stage 1 and stage 2 attributes, EL1&0 translation
       regime'.

   •   The encoding of the stage 2 memory type and cacheability attributes is derived
       from MAIR_EL2 register as described in the Armv8-R AArch64 architecture.

0b1   When this bit is 1, then:

   •   The inner and outer memory attributes for stage 2 EL1&0 translation regime
       must be the same with the same encoding, otherwise, the combined attribute is
       UNKNOWN.

   •   If stage 2 EL1&0 translation regime memory attribute is Write-Back with
       MAIR_EL2.Attr[7:6] = 0b11, then the combined attribute is Normal Write-Back.
       For all other encodings, the combination of stage 1 and stage 2 translations on
       memory type and cacheability attributes are as described in the Armv8.0
       architecture.

This bit is permitted to be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [45:42]

Reserved, RES0.

API, bit [41]

When FEAT_PAuth is implemented:

Controls the use of instructions related to Pointer Authentication:

- In EL0, when HCR_EL2.TGE==0 or HCR_EL2.E2H==0, and the associated SCTLR_EL1.En<N><M>==1.

- In EL1, the associated SCTLR_EL1.En<N><M>==1.

Traps are reported using EC syndrome value 0x09. The Pointer Authentication instructions trapped are:

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB.

- PACGA, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB.

- RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ.

- ERETAA, ERETAB, LDRAA and LDRAB.

0b0   The instructions related to Pointer Authentication are trapped to EL2, when EL2 is enabled in the current Security state and the instructions are enabled for the EL1&0 translation regime, from:

- EL0 when HCR_EL2.TGE==0 or HCR_EL2.E2H==0.

- EL1.

If HCR_EL2.NV is 1, the HCR_EL2.NV trap takes precedence over the HCR_EL2.API trap for the ERETAA and ERETAB instructions.

If EL2 is implemented and enabled in the current Security state and HFGITR_EL2.ERET == 1, execution at EL1 using AArch64 of ERETAA or ERETAB instructions is reported with EC syndrome value 0x1A with its associated ISS field, as the fine-grained trap has higher priority than the HCR_EL2.API == 0.

0b1   This control does not cause any instructions to be trapped.

If FEAT_PAuth is implemented but EL2 is not implemented or disabled in the current Security state, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APK, bit [40]

When FEAT_PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers from EL1 to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value 0x18:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1, APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1, APGAKeyLo_EL1, and APGAKeyHi_EL1.

0b0   Access to the registers holding "key" values for pointer authentication from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.

0b1   This control does not cause any instructions to be trapped.

———— **Note** ————

If FEAT_PAuth is implemented but EL2 is not implemented or is disabled in the current Security state, the system behaves as if this bit is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [39]

Reserved, RES0.

MIOCNCE, bit [38]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the EL1&0 translation regimes.

0b0	For the EL1&0 translation regimes, 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.
0b1	For the EL1&0 translation regimes, 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'.

This field can be implemented as RAZ/WI.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

TEA, bit [37]

When FEAT_RAS is implemented:

Route synchronous External abort exceptions to EL2.

0b0	This control does not cause exceptions to be routed from EL0 and EL1 to EL2.
0b1	Route synchronous External abort exceptions from EL0 and EL1 to EL2, when EL2 is enabled in the current Security state, if not routed to EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TERR, bit [36]

When FEAT_RAS is implemented:

Trap Error record accesses. Trap accesses to the RAS error registers from EL1 to EL2 as follows:

• If EL1 is using AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:

— ERRIDR_EL1, ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXFR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.

— When FEAT_RASv1p1 is implemented, ERXMISC2_EL1, and ERXMISC3_EL1.

0b0	This control does not cause any instructions to be trapped.
0b1	Accesses to the specified registers from EL1 generate a Trap exception to EL2, when EL2 is enabled in the current Security state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:34]

Reserved, RES0.

ID, bit [33]

Stage 2 Instruction access cacheability disable. For the EL1&0 translation regime, when EL2 is
enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2
translations for instruction accesses to Normal memory to be Non-cacheable.

———— **Note** ————

The behavior is same irrespective of whether the instruction accesses is to MPU region or
Backgroud Region.

————————————

0b0 This control has no effect on stage 2 of the EL1&0 translation regime.

0b1 Forces all stage 2 translations for instruction accesses to Normal memory to be
    Non-cacheable.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE
ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

CD, bit [32]

Stage 2 Data access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled
in the current Security state and 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.

———— **Note** ————

The behavior is same irrespective of whether the data accesses is to MPU region or Backgroud
Region.

————————————

0b0 This control has no effect on stage 2 of the EL1&0 translation regime for data accesses
    and translation table walks.

0b1 Forces all stage 2 translations for data accesses and translation table walks to Normal
    memory to be Non-cacheable.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE
ignores the value of this field for all purposes other than a direct read of this field.

This field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RAO/WI.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps EL1 reads of the virtual memory control registers to
EL2, when EL2 is enabled in the current Security state, as follows:

• If EL1 is using AArch64 state, the following registers are trapped to EL2 and reported using
 EC syndrome value 0x18.

  — SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1,
   AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.

     — If EL1 is in PMSAv8-64 context, the following registers are also trapped to EL2 and reported using EC syndrome value `0x18` - PRENR_EL1, PRSELR_EL1, PRBAR_EL1, PRBAR<n>_EL1, PRLAR_EL1, PRLAR<n>_EL1

    `0b0`   This control does not cause any instructions to be trapped.

    `0b1`   EL1 read accesses to the specified Virtual Memory controls are trapped to EL2, when EL2 is enabled in the current Security state.

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.

--- **Note** ---

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the memory system. A hypervisor might use this trap as part of its virtualization of memory management.

This field resets to an architecturally UNKNOWN value.

HCD, bit [29]

HVC instruction disable. Disables EL1 execution of HVC instructions, from both Execution states, when EL2 is enabled in the current Security state, reported using EC syndrome value `0x00`.

    `0b0`   HVC instruction execution is enabled at EL2 and EL1.

    `0b1`   HVC instructions are UNDEFINED at EL2 and 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 field resets to an architecturally UNKNOWN value.

TDZ, bit [28]

Trap DC ZVA instructions. Traps EL0 and EL1 execution of DC ZVA instructions to EL2, when EL2 is enabled in the current Security state, from AArch64 state only, reported using EC syndrome value `0x18`.

    `0b0`   This control does not cause any instructions to be trapped.

    `0b1`   In AArch64 state, any attempt to execute an instruction this trap applies to at EL1, or at EL0 when the instruction is not UNDEFINED at EL0, is trapped to EL2 when EL2 is enabled in the current Security state.

             Reading the DCZID_EL0 returns a value that indicates that the instructions this trap applies to are not supported.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TGE, bit [27]

Trap General Exceptions, from EL0.

    `0b0`   This control has no effect on execution at EL0.

    `0b1`   When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0.

             When EL2 is enabled in the current Security state, in all cases:

           • All exceptions that would be routed to EL1 are routed to EL2.

           • If EL1 is using AArch64, the SCTLR_EL1.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1.

- If stage 1 EL1&0 translation regime is in PMSAv8-64 context, the SCTLR_EL1.BR field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1.

- 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.

- The MDCR_EL2.{TDRA, TDOSA, TDA, TDE} fields are treated as being 1 for all purposes other than returning the result of a direct read of MDCR_EL2.

In addition, when EL2 is enabled in the current Security state, if:

- HCR_EL2.E2H is 0, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 1.

- HCR_EL2.E2H is 1, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 0.

For further information on the behavior of this bit when E2H is 1, see 'Behavior of HCR_EL2.E2H'.

HCR_EL2.TGE must not be cached in a TLB.

This field resets to an architecturally UNKNOWN value.

TVM, bit [26]

Trap Virtual Memory controls. Traps EL1 writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, the following registers are trapped to EL2 and reported using EC syndrome value 0x18:

  — SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.

  — If EL1 is in PMSAv8-64 context, the following registers are also trapped to EL2 and reported using EC syndrome value 0x18 - PRENR_EL1, PRSELR_EL1, PRBAR_EL1, PRBAR<n>_EL1, PRLAR_EL1, PRLAR<n>_EL1

0b0   This control does not cause any instructions to be trapped.

0b1   EL1 write accesses to the specified EL1 virtual memory control registers are trapped to EL2, when EL2 is enabled in the current Security state.

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.

This field resets to an architecturally UNKNOWN value.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps EL1 execution of TLB maintenance instructions to EL2, when EL2 is enabled in the current Security state, as follows:

- When EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported using EC syndrome value 0x18:

  — TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1.

  — TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS.

  — If FEAT_TLBIOS is implemented, this trap applies to TLBI VMALLE1OS, TLBI VAE1OS, TLBI ASIDE1OS, TLBI VAAE1OS, TLBI VALE1OS, TLBI VAALE1OS.

  — If FEAT_TLBIRANGE is implemented, this trap applies to TLBI RVAE1, TLBI RVAAE1, TLBI RVALE1, TLBI RVAALE1, TLBI RVAE1IS, TLBI RVAAE1IS, TLBI RVALE1IS, TLBI RVAALE1IS.

  — If FEAT_TLBIOS and FEAT_TLBIRANGE are implemented, this trap apppplies to TLBI RVAE1OS, TLBI RVAAE1OS, TLBI RVALE1OS, TLBI RVAALE1OS.

0b0   This control does not cause any instructions to be trapped.

0b1   EL1 execution of the specified TLB maintenance instructions are trapped to EL2, when EL2 is enabled in the current Security state.

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.

  —— **Note** ——

The TLB maintenance instructions are UNDEFINED at EL0.

This field resets to an architecturally UNKNOWN value.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, the following instructions are trapped to EL2 and reported with EC syndrome value 0x18:

  — IC IVAU, DC CVAU. 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.

- If EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported with EC syndrome value 0x18:

  — IC IVAU, IC IALLU, IC IALLUIS, DC CVAU.

  —— **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.

0b0   This control does not cause any instructions to be trapped.

0b1   Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TPCP, bit [23]

When FEAT_DPB is implemented:

Trap data or unified cache maintenance instructions that operate to the Point of Coherency or Persistence. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, the following instructions are trapped to EL2 and reported using EC syndrome value 0x18:

  — DC CIVAC, DC CVAC, DC CVAP. 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.

- If EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported using EC syndrome value `0x18`:

  — DC IVAC, DC CIVAC, DC CVAC, DC CVAP.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

———— **Note** ————

- An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

  — AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.

- In Armv8.0 and Armv8.1, this field is named TPC. From Armv8.2 it is named TPCP.

———————————————

`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 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.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, accesses to the following registers are trapped and reported using EC syndrome value `0x18`:

  — 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.

- If EL1 is using AArch64 state, accesses to DC IVAC, DC CIVAC, DC CVAC are trapped and reported using EC syndrome value `0x18`.

———— **Note** ————

- An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

  — AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.

- In Armv8.0 and Armv8.1, this field is named TPC. From Armv8.2 it is named TPCP.

———————————————

`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64 state, accesses to DC ISW, DC CSW, DC CISW are trapped to EL2, reported using EC syndrome value `0x18`.

—— **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.

`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

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.

This field resets to an architecturally UNKNOWN value.

TACR, bit [21]

Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, accesses to ACTLR_EL1 to EL2, are trapped to EL2 and reported using EC syndrome value `0x18`.

`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to the specified registers are trapped to EL2, when EL2 is enabled in the current Security state.

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.

—— **Note** ——

ACTLR_EL1 is not accessible at EL0

The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

This field resets to an architecturally UNKNOWN value.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2, when EL2 is enabled in the current Security state as follows:

- In AArch64 state, access to any of the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome `0x18`:
  — IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}.
  — IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.

When the value of HCR_EL2.TIDCP is 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from EL0 is trapped to EL2. If it is not, then it is UNDEFINED, and any attempt to access it from EL0 generates an exception that is taken to EL1.

`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2, when EL2 is enabled in the current Security state.

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

———— **Note** ————

Arm expects the trapping of EL0 accesses to these functions to EL2 to be unusual, and used only when the hypervisor is virtualizing EL0 operation. Arm strongly recommends that unless the hypervisor must virtualize EL0 operation, an EL0 access to any of these functions is UNDEFINED, as it would be if the implementation did not include EL2. The PE then takes any resulting exception to EL1.

The trapping of accesses to these registers from EL1 is higher priority than an exception resulting from the register access being UNDEFINED.

————————————

This field resets to an architecturally UNKNOWN value.

TSC, bit [19]

Trap SMC instructions. Traps EL1 execution of SMC instructions to EL2, when EL2 is enabled in the current Security state.

If execution is in AArch64 state the trap is reported using EC syndrome value 0x17.

———— **Note** ————

HCR_EL2.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Trap exceptions and SMC exceptions have different preferred return addresses.

————————————

0b0   This control does not cause any instructions to be trapped.

0b1   If EL3 is not implemented, FEAT_NV is implemented, and HCR_EL2.NV is 1, then any attempt to execute an SMC instruction at EL1 using AArch64 is trapped to EL2, when EL2 is enabled in the current Security state.

        If EL3 is not implemented, and either FEAT_NV is not implemented or HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether:

        • Any attempt to execute an SMC instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state.

        • Any attempt to execute an SMC instruction is UNDEFINED.

SMC instructions are UNDEFINED at EL0.

If EL3 is not implemented and HCR_EL2.NV is 0, it is IMPLEMENTATION DEFINED whether this bit is:

• RES0.

• Implemented with the functionality as described in HCR_EL2.TSC.

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.

This field resets to an architecturally UNKNOWN value.

TID3, bit [18]

Trap ID group 3. Traps EL1 reads of group 3 ID registers to EL2, when EL2 is enabled in the current Security state, as follows:

In AArch64 state:

• Reads of the following registers are trapped to EL2, reported using EC syndrome value 0x18:

        — ID_PFR0_EL1, ID_PFR1_EL1, ID_PFR2_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.

> — ID_MMFR4_EL1 and ID_MMFR5_EL1 are trapped to EL2, unless implemented as
> RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4_EL1 or
> ID_MMFR5_EL1 are trapped to EL2.

> — ID_AA64MMFR2_EL1 and ID_ISAR6_EL1 are trapped to EL2, unless implemented
> as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to
> ID_AA64MMFR2_EL1 and ID_ISAR6_EL1 are trapped to EL2.

> — Otherwise, it is IMPLEMENTATION DEFINED whether this field traps MRS accesses to
> registers 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}.

0b0   This control does not cause any instructions to be trapped.

0b1   The specified EL1 read accesses to ID group 3 registers are trapped to EL2, when EL2 is enabled in the current Security state.

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.

This field resets to an architecturally UNKNOWN value.

TID2, bit [17]

Trap ID group 2. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64, reads of CTR_EL0, CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1 are trapped to EL2, reported using EC syndrome value 0x18.

- If EL0 is using AArch64 and the value of SCTLR_EL1.UCT is not 0, reads of CTR_EL0 are trapped to EL2, reported using EC syndrome value 0x18. If the value of SCTLR_EL1.UCT is 0 then EL0 reads of CTR_EL0 are UNDEFINED and any resulting exception takes precedence over this trap.

- If EL1 is using AArch64, writes to CSSELR_EL1 are trapped to EL2, reported using EC syndrome value 0x18.

0b0   This control does not cause any instructions to be trapped.

0b1   The specified EL1 and EL0 accesses to ID group 2 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

TID1, bit [16]

Trap ID group 1. Traps EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state as follows:

- Accesses of MPUIR_EL1, REVIDR_EL1, AIDR_EL1, reported using EC syndrome value 0x18.

0b0   This control does not cause any instructions to be trapped.

0b1   The specified EL1 read accesses to ID group 1 registers are trapped to EL2, when EL2 is enabled in the current Security state.

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.

This field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

TWE, bit [14]

Traps EL0 and EL1 execution of WFE instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value 0x01.

0b0   This control does not cause any instructions to be trapped.

0b1   Any attempt to execute a WFE instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE or SCTLR_EL1.nTWE.

---- **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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information about when WFE instructions can cause the PE to enter a low-power state, see 'Wait for Event mechanism and Send event'.

This field resets to an architecturally UNKNOWN value.

TWI, bit [13]

Traps EL0 and EL1 execution of WFI instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value 0x01.

0b0   This control does not cause any instructions to be trapped.

0b1   Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI or SCTLR_EL1.nTWI.

---- **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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information about when WFI instructions can cause the PE to enter a low-power state, see 'Wait for Interrupt'.

This field resets to an architecturally UNKNOWN value.

DC, bit [12]

Default Cacheability.

0b0   This control has no effect on the EL1&0 translation regime.

0b1   In both Security states:

- 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.

- If stage 1 EL1&0 translation regime is in PMSAv8-64 context, the PE behaves as if the value of the SCTLR_EL1.BR field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1.

- 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-Allocate Write-Allocate.

This field has no effect on the EL2, EL2&0, and EL3 translation regimes.

This field is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this field.

This field resets to an architecturally UNKNOWN value.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from EL1 or EL0:

0b00   No effect.

0b01   Inner Shareable.

0b10   Outer Shareable.

0b11   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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0b00 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

FB, bit [9]

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from EL1:

AArch64: TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1, IC IALLU, TLBI RVAE1, TLBI RVAAE1, TLBI RVALE1, TLBI RVAALE1.

0b0   This field has no effect on the operation of the specified instructions.

0b1   When one of the specified instruction is executed at EL1, the instruction is broadcast within the Inner Shareable shareability domain.

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.

This field resets to an architecturally UNKNOWN value.

VSE, bit [8]

Virtual SError interrupt.

0b0   This mechanism is not making a virtual SError interrupt pending.

0b1   A virtual SError interrupt is pending because of this mechanism.

The virtual SError interrupt is only enabled when the value of HCR_EL2.{TGE, AMO} is {0, 1}.

This field resets to an architecturally UNKNOWN value.

VI, bit [7]

Virtual IRQ Interrupt.

0b0   This mechanism is not making a virtual IRQ pending.

0b1   A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR_EL2.{TGE, IMO} is {0, 1}.

This field resets to an architecturally UNKNOWN value.

VF, bit [6]

Virtual FIQ Interrupt.

0b0 This mechanism is not making a virtual FIQ pending.

0b1 A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR_EL2.{TGE, FMO} is {0, 1}.

This field resets to an architecturally UNKNOWN value.

AMO, bit [5]

Physical SError interrupt routing.

0b0 When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:

- Physical SError interrupts are not taken to EL2.

- When the value of HCR_EL2.TGE is 0, if the PE is executing at EL2 using AArch64, physical SError interrupts are not taken unless they are routed to EL3 by the SCR_EL3.EA bit.

- Virtual SError interrupts are disabled.

0b1 When executing at any Exception level, and EL2 is enabled in the current Security state:

- Physical SError interrupts are taken to EL2, unless they are routed to EL3.

- When the value of HCR_EL2.TGE is 0, then virtual SError interrupts are enabled.

If EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1:

- Regardless of the value of the AMO bit physical asynchronous External aborts and SError interrupts target EL2 unless they are routed to EL3.

- When FEAT_VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.

- When FEAT_VHE is implemented and 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'.

This field resets to an architecturally UNKNOWN value.

IMO, bit [4]

Physical IRQ Routing.

0b0 When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:

- Physical IRQ interrupts are not taken to EL2.

- When the value of HCR_EL2.TGE is 0, if the PE is 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.

0b1 When executing at any Exception level, and EL2 is enabled in the current Security state:

- Physical IRQ interrupts are taken to EL2, unless they are routed to EL3.

- When the value of HCR_EL2.TGE is 0, then Virtual IRQ interrupts are enabled.

If EL2 is enabled in the current Security state, and the value of HCR_EL2.TGE is 1:

- Regardless of the value of the IMO bit, physical IRQ Interrupts target EL2 unless they are routed to EL3.

- When FEAT_VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.

- When FEAT_VHE is implemented and 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'.

This field resets to an architecturally UNKNOWN value.

FMO, bit [3]

Physical FIQ Routing.

0b0   When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:

- Physical FIQ interrupts are not taken to EL2.

- When the value of HCR_EL2.TGE is 0, if the PE is 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.

0b1   When executing at any Exception level, and EL2 is enabled in the current Security state:

- Physical FIQ interrupts are taken to EL2, unless they are routed to EL3.

- When HCR_EL2.TGE is 0, then Virtual FIQ interrupts are enabled.

If EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1:

- Regardless of the value of the FMO bit, physical FIQ Interrupts target EL2 unless they are routed to EL3.

- When FEAT_VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.

- When FEAT_VHE is implemented and 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'.

This field resets to an architecturally UNKNOWN value.

PTW, bit [2]

Protected Table Walk. In the 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:

0b0   The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.

0b1   The memory access generates a stage 2 Permission fault.

This field is permitted to be cached in a TLB.

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.

This field resets to an architecturally UNKNOWN value.

SWIO, bit [1]

Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way:

0b0   This control has no effect on the operation of data cache invalidate by set/way instructions.

0b1   Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When the value of this bit is 1:

AArch64: DC ISW performs the same invalidation as a DC CISW instruction.

This bit can be implemented as RES1.

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.

This field resets to an architecturally UNKNOWN value.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the EL1&0 translation regime, when EL2 is enabled in the current Security state.

0b0   EL1&0 stage 2 address translation disabled.

0b1   EL1&0 stage 2 address translation enabled.

If HCR_EL2.VM is 1 and SCTLR_EL2.{M, BR} is {0, 0}, then the behavior is a CONSTRAINED UNPREDICTABLE choice of:

• The memory attribute becomes UNKNOWN.

• Raise stage-2 level 0 translation fault.

When the value of this bit is 1, data cache invalidate instructions executed at EL1 perform a data cache clean and invalidate. 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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

This field resets to an architecturally UNKNOWN value.

Accessing the HCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, HCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return HCR_EL2;
```

MSR HCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    HCR_EL2 = X[t];
```

G1.3.4   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 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

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:

63 44	43 40	39 36	35 32	31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
RES0	TraceFilt	DoubleLock	RES0	CTX_CMPs	RES0	WRPs	RES0	BRPs	PMUVer	TraceVer	DebugVer

Bits [63:44]

Reserved, RES0.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version. Defined values are:

0b0000   Armv8.4 Self-hosted Trace Extension not implemented.

0b0001   Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if an Embedded Trace Macrocell Architecture PE Trace Unit is implemented, the value 0b0000 is not permitted.

DoubleLock, bits [39:36]

OS Double Lock implemented. Defined values are:

0b0000   OS Double Lock implemented. OSDLR_EL1 is RW.

0b1111   OS Double Lock not implemented. OSDLR_EL1 is RAZ/WI.

All other values are reserved.

FEAT_DoubleLock implements the functionality identified by the value 0b0000.

In Armv8.0, the only permitted value is 0b0000.

If FEAT_Debugv8p2 is implemented and FEAT_DoPD is not implemented, the permitted values are 0b0000 and 0b1111.

If FEAT_DoPD is implemented, the only permitted value is 0b1111.

Bits [35:32]

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.
>
> This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'
>
> Defined values are:
>
> `0b0000` Performance Monitors Extension not implemented.
>
> `0b0001` Performance Monitors Extension, PMUv3 implemented.
>
> `0b0100` PMUv3 for Armv8.1. As `0b0001`, and also includes support for:
>
> > • Extended 16-bit PMEVTYPER<n>_EL0.evtCount field.
> >
> > • If EL2 is implemented, the MDCR_EL2.HPMD control bit.
>
> `0b0101` PMUv3 for Armv8.4. As `0b0100`, and also includes support for the PMMIR_EL1 register.
>
> `0b1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.
>
> All other values are reserved.
>
> FEAT_PMUv3 implements the functionality identified by the value `0b0001`.
>
> FEAT_PMUv3p1 implements the functionality identified by the value `0b0100`.
>
> FEAT_PMUv3p4 implements the functionality identified by the value `0b0101`.
>
> From Armv8.1, if FEAT_PMUv3 is implemented, the value `0b0001` is not permitted.
>
> From Armv8.4, if FEAT_PMUv3 is implemented, the value `0b0100` is not permitted.

TraceVer, bits [7:4]

> Trace support. Indicates whether System register interface to a PE trace unit is implemented. Defined values are:
>
> `0b0000` PE trace unit System registers not implemented.
>
> `0b0001` PE trace unit System registers implemented.
>
> All other values are reserved.
>
> See the ETM Architecture Specification for more information.
>
> A value of `0b0000` only indicates that no System register interface to a PE trace unit is implemented. A PE trace unit might nevertheless be implemented without a System register interface.

DebugVer, bits [3:0]

> Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:
>
> `0b0110` Armv8 debug architecture.
>
> `0b0111` Armv8 debug architecture with Virtualization Host Extensions.
>
> `0b1000` Armv8.2 debug architecture.
>
> `0b1001` Armv8.4 debug architecture.

All other values are reserved.

FEAT_Debugv8p2 adds the functionality identified by the value `0b1000`.

FEAT_Debugv8p4 adds the functionality identified by the value `0b1001`.

In Armv8.1, the value `0b0110` is not permitted.

In Armv8.2, the value `0b0111` is not permitted.

From Armv8.4, the value `0b1000` is not permitted.

Accessing the ID_AA64DFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64DFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64DFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64DFR0_EL1;
```

G1.3.5 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.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.
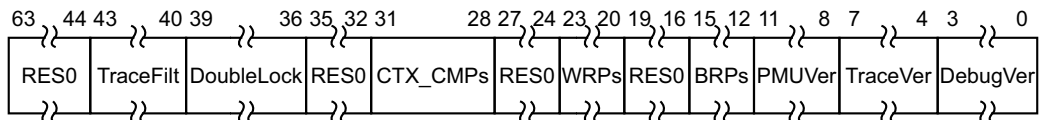
Configurations

There are no configuration notes.

Attributes

ID_AA64ISAR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR0_EL1 bit assignments are:

63 60	59 56	55 52	51 48	47 44	43 40	39 36	35 32	31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
RES0	TLB	TS	FHM	DP	SM4	SM3	SHA3	RDM	RES0	Atomic	CRC32	SHA2	SHA1	AES	RES0
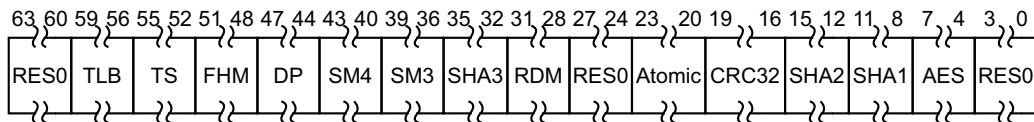
Bits [63:60]

Reserved, RES0.

TLB, bits [59:56]

Indicates support for Outer shareable and TLB range maintenance instructions. Defined values are:

0b0000   Outer shareable and TLB range maintenance instructions are not implemented.

0b0001   Outer shareable TLB maintenance instructions are implemented.

0b0010   Outer shareable and TLB range maintenance instructions are implemented.

All other values are reserved.

FEAT_TLBIOS implements the functionality identified by the values 0b0001 and 0b0010.

FEAT_TLBIRANGE implements the functionality identified by the value 0b0010.

From Armv8.4, the only permitted value is 0b0010.

TS, bits [55:52]

Indicates support for flag manipulation instructions. Defined values are:

0b0000   No flag manipulation instructions are implemented.

0b0001   CFINV, RMIF, SETF16, and SETF8 instructions are implemented.

All other values are reserved.

FEAT_FlagM implements the functionality identified by the value 0b0001.

In Armv8.2, the permitted values are 0b0000 and 0b0001.

In Armv8.4, the only permitted value is 0b0001.

FHM, bits [51:48]

Indicates support for FMLAL and FMLSL instructions. Defined values are:

0b0000   FMLAL and FMLSL instructions are not implemented.

0b0001   FMLAL and FMLSL instructions are implemented.

All other values are reserved.

FEAT_FHM implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

DP, bits [47:44]

Indicates support for Dot Product instructions in AArch64 state. Defined values are:

0b0000   No Dot Product instructions implemented.

0b0001   UDOT and SDOT instructions implemented.

All other values are reserved.

FEAT_DotProd implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

SM4, bits [43:40]

Indicates support for SM4 instructions in AArch64 state. Defined values are:

0b0000   No SM4 instructions implemented.

0b0001   SM4E and SM4EKEY instructions implemented.

All other values are reserved.

If FEAT_SM4 is not implemented, the value 0b0001 is reserved.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM3.

SM3, bits [39:36]

Indicates support for SM3 instructions in AArch64 state. Defined values are:

0b0000   No SM3 instructions implemented.

0b0001   SM3SS1, SM3TT1A, SM3TT1B, SM3TT2A, SM3TT2B, SM3PARTW1, and SM3PARTW2 instructions implemented.

All other values are reserved.

If FEAT_SM3 is not implemented, the value 0b0001 is reserved.

FEAT_SM3 implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM4.

SHA3, bits [35:32]

Indicates support for SHA3 instructions in AArch64 state. Defined values are:

0b0000   No SHA3 instructions implemented.

0b0001   EOR3, RAX1, XAR, and BCAX instructions implemented.

All other values are reserved.

If FEAT_SHA3 is not implemented, the value 0b0001 is reserved.

FEAT_SHA3 implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR0_EL1.SHA1 is 0b0000, this field must have the value 0b0000.

If the value of this field is 0b0001, ID_AA64ISAR0_EL1.SHA2 must have the value 0b0010.

RDM, bits [31:28]

Indicates support for SQRDMLAH and SQRDMLSH instructions in AArch64 state. Defined values are:

0b0000   No RDMA instructions implemented.

0b0001   SQRDMLAH and SQRDMLSH instructions implemented.

All other values are reserved.

        FEAT_RDM implements the functionality identified by the value 0b0001.

        From Armv8.1, the only permitted value is 0b0001.

Bits [27:24]

        Reserved, RES0.

Atomic, bits [23:20]

        Indicates support for Atomic instructions in AArch64 state. Defined values are:

0b0000	No Atomic instructions implemented.
0b0010	LDADD, LDCLR, LDEOR, LDSET, LDSMAX, LDSMIN, LDUMAX, LDUMIN, CAS, CASP, and SWP instructions implemented.

        All other values are reserved.

        FEAT_LSE implements the functionality identified by the value 0b0010.

        From Armv8.1, the only permitted value is 0b0010.

CRC32, bits [19:16]

        Indicates support for CRC32 instructions in AArch64 state. Defined values are:

0b0000	No CRC32 instructions implemented.
0b0001	CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX instructions implemented.

        All other values are reserved.

        In Armv8.0, the permitted values are 0b0000 and 0b0001.

        From Armv8.1, the only permitted value is 0b0001.

SHA2, bits [15:12]

        Indicates support for SHA2 instructions in AArch64 state. Defined values are:

0b0000	No SHA2 instructions implemented.
0b0001	Implements instructions: SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
0b0010	Implements instructions:

-     SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
-     SHA512H, SHA512H2, SHA512SU0, and SHA512SU1.

        All other values are reserved.

        FEAT_SHA256 implements the functionality identified by the value 0b0001.

        FEAT_SHA512 implements the functionality identified by the value 0b0010.

        In Armv8, the permitted values are 0b0000 and 0b0001.

        From Armv8.2, the permitted values are 0b0000, 0b0001, and 0b0010.

        If the value of ID_AA64ISAR0_EL1.SHA1 is 0b0000, this field must have the value 0b0000.

        If the value of this field is 0b0010, ID_AA64ISAR0_EL1.SHA3 must have the value 0b0001.

SHA1, bits [11:8]

        Indicates support for SHA1 instructions in AArch64 state. Defined values are:

0b0000	No SHA1 instructions implemented.
0b0001	SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 instructions implemented.

        All other values are reserved.

        FEAT_SHA1 implements the functionality identified by the value 0b0001.

        From Armv8, the permitted values are 0b0000 and 0b0001.

        If the value of ID_AA64ISAR0_EL1.SHA2 is 0b0000, this field must have the value 0b0000.

AES, bits [7:4]

Indicates support for AES instructions in AArch64 state. Defined values are:

0b0000   No AES instructions implemented.

0b0001   AESE, AESD, AESMC, and AESIMC instructions implemented.

0b0010   As for 0b0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

From Armv8, the permitted values are 0b0000 and 0b0010.

Bits [3:0]

Reserved, RES0.

Accessing the ID_AA64ISAR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64ISAR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64ISAR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64ISAR0_EL1;
```

G1.3.6 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

Configurations

If ID_AA64ISAR1_EL1.{API, APA} == {0000, 0000}, then:

- The TCR_EL1.{TBID,TBID0}, TCR_EL2.{TBID0,TBID1}, TCR_EL2.TBID and TCR_EL3.TBID bits are RES0.

- APIAKeyHi_EL1, APIAKeyLo_EL1, APIBKeyHi_EL1, APIBKeyLo_EL1, APDAKeyHi_EL1, APDAKeyLo_EL1, APDBKeyHi_EL1, APDBKeyLo_EL1 are not allocated.

- SCTLR_ELx.EnIA, SCTLR_ELx.EnIB, SCTLR_ELx.EnDA, SCTLR_ELx.EnDB are all RES0.

If ID_AA64ISAR1_EL1.{GPI, GPA, API, APA} == {0000, 0000, 0000, 0000}, then:

- HCR_EL2.APK and HCR_EL2.API are RES0.

- SCR_EL3.APK and SCR_EL3.API are RES0.

Attributes

ID_AA64ISAR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR1_EL1 bit assignments are:
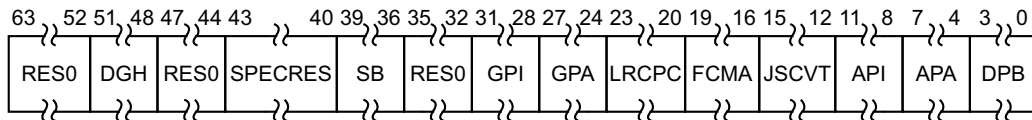
Bits [63:52]

Reserved, RES0.

DGH, bits [51:48]

Indicates support for the Data Gathering Hint instruction. Defined values are:

0b0000   Data Gathering Hint is not implemented.

0b0001   Data Gathering Hint is implemented.

All other values are reserved.

FEAT_DGH implements the functionality identified by 0b0001.

From ARMv8.0, the permitted values are 0b0000 and 0b0001.

If the DGH instruction has no effect in preventing the merging of memory accesses, the value of this field is 0b0000.

Bits [47:44]

Reserved, RES0.

SPECRES, bits [43:40]

Indicates support for prediction invalidation instructions in AArch64 state. Defined values are:

0b0000   CFP RCTX, DVP RCTX, and CPP RCTX instructions are not implemented.

0b0001   CFP RCTX, DVP RCTX, and CPP RCTX instructions are implemented.

All other values are reserved.

FEAT_PredInv implements the functionality identified by 0b0001.

In Armv8-R, the only permitted value is 0b0001.

SB, bits [39:36]

Indicates support for SB instruction in AArch64 state. Defined values are:

0b0000   SB instruction is not implemented.

0b0001   SB instruction is implemented.

All other values are reserved.

FEAT_SB implements the functionality identified by 0b0001.

In Armv8-R, the only permitted value is 0b0001.

Bits [35:32]

Reserved, RES0.

GPI, bits [31:28]

Indicates support for an IMPLEMENTATION DEFINED algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

0b0000   Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.

0b0001   Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPA is non-zero, this field must have the value 0b0000.

GPA, bits [27:24]

Indicates whether QARMA or Architected algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

0b0000   Generic Authentication using an Architected algorithm is not implemented.

0b0001   Generic Authentication using the QARMA algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is non-zero, this field must have the value 0b0000.

LRCPC, bits [23:20]

Indicates support for weaker release consistency, RCpc, based model. Defined values are:

0b0000   The LDAPR*, LDAPUR*, and STLUR* instructions are not implemented.

0b0001   The LDAPR* instructions are implemented.
          The LDAPUR*, and STLUR* instructions are not implemented.

0b0010   The LDAPR*, LDAPUR*, and STLUR* instructions are implemented.

All other values are reserved.

FEAT_LRCPC implements the functionality identified by the value 0b0001.

FEAT_LRCPC2 implements the functionality identified by the value 0b0010.

In Armv8.2, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

In Armv8.3, the permitted values are `0b0001` and `0b0010`.

From Armv8.4, the only permitted value is `0b0010`.

FCMA, bits [19:16]

Indicates support for complex number addition and multiplication, where numbers are stored in vectors. Defined values are:

`0b0000`   The FCMLA and FCADD instructions are not implemented.

`0b0001`   The FCMLA and FCADD instructions are implemented.

All other values are reserved.

FEAT_FCMA implements the functionality identified by the value `0b0001`.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is `0b0000`.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is `0b0001`.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is `0b0000`.

JSCVT, bits [15:12]

Indicates support for JavaScript conversion from double precision floating point values to integers in AArch64 state. Defined values are:

`0b0000`   The FJCVTZS instruction is not implemented.

`0b0001`   The FJCVTZS instruction is implemented.

All other values are reserved.

FEAT_JSCVT implements the functionality identified by `0b0001`.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is `0b0000`.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is `0b0001`.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is `0b0000`.

API, bits [11:8]

Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

`0b0000`   Address Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.

`0b0001`   Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.

`0b0010`   Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE, and the HaveEnhancedPAC2() function returning FALSE.

`0b0011`   Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

`0b0100`   Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.

0b0101   Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality added by the values 0b0000, 0b0001, and 0b0010.

FEAT_PAuth2 implements the functionality added by the value 0b0011.

FEAT_FPAC implements the functionality added by the values 0b0100 and 0b0101.

From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.APA is non-zero, this field must have the value 0b0000.

APA, bits [7:4]

Indicates whether QARMA or Architected algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

0b0000   Address Authentication using an Architected algorithm is not implemented.

0b0001   Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.

0b0010   Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE and the HaveEnhancedPAC2() function returning FALSE.

0b0011   Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning FALSE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.

0b0100   Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.

0b0101   Address Authentication using the QARMA algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality added by the values 0b0000, 0b0001, and 0b0010.

FEAT_PAuth2 implements the functionality added by the value 0b0011.

FEAT_FPAC implements the functionality added by the values 0b0100 and 0b0101.

From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of the ID_AA64ISAR1_EL1.API is non-zero, this field must have the value 0b0000.

DPB, bits [3:0]

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state. Defined values are:

0b0000   DC CVAP not supported.

0b0001   DC CVAP supported.

0b0010   DC CVAP and DC CVADP supported.

All other values are reserved.

FEAT_DPB implements the functionality identified by the value 0b0001.

FEAT_DPB2 implements the functionality identified by the value 0b0010.

In Armv8.2, the permitted values are 0b0001 and 0b0010.

In Armv8-R, the only permitted value is `0b0010`

Accessing the ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64ISAR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64ISAR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64ISAR1_EL1;
```

G1.3.7 ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0

The ID_AA64MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

Configurations

There are no configuration notes.

Attributes

ID_AA64MMFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR0_EL1 bit assignments are:
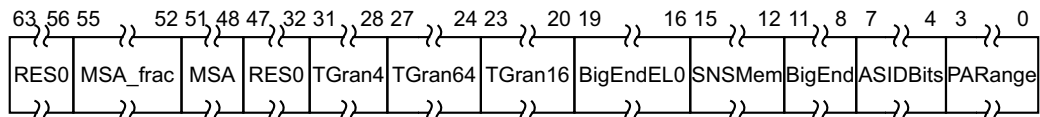
Bits [63:56]

Reserved, RES0.

MSA_frac, bits [55:52]

Memory System Architecture fractional field. This holds the information on additional Memory System Architectures supported. Defined values are:

0b0000   PMSAv8-64 not supported in any translation regime.

0b0001   PMSAv8-64 supported in all translation regimes. No support for VMSAv8-64.

0b0010   PMSAv8-64 supported in all translation regimes. In addition to PMSAv8-64, stage 1 EL1&0 translation regime also supports VMSAv8-64.

All other values are reserved.

The permitted values are 0b0001 and 0b0010.

This field is valid only when ID_AA64MMFR0_EL1.MSA is 0b1111.

MSA, bits [51:48]

Memory System Architecture ID field. This holds the information on Memory System Architectures supported. Defined values are:

0b0000   VMSAv8-64 supported in all translation regimes. No support for PMSAv8-64.

0b1111   See ID_AA64MMFR0_EL1.MSA_frac for the Memory System Architectures supported.

All other values are reserved.

In Armv8-R, the only permitted value is 0b1111.

Bits [47:32]

Reserved, RES0.

TGran4, bits [31:28]

Indicates support for 4KB memory translation granule size. Defined values are:

0b0000   4KB granule supported.

0b1111   4KB granule not supported.

All other values are reserved.

TGran64, bits [27:24]

Indicates support for 64KB memory translation granule size. Defined values are:

0b0000   64KB granule supported.

0b1111   64KB granule not supported.

All other values are reserved.

TGran16, bits [23:20]

Indicates support for 16KB memory translation granule size. Defined values are:

0b0000   16KB granule not supported.

0b0001   16KB granule supported.

All other values are reserved.

BigEndEL0, bits [19:16]

Indicates support for mixed-endian at EL0 only. Defined values are:

0b0000   No mixed-endian support at EL0. The SCTLR_EL1.E0E bit has a fixed value.

0b0001   Mixed-endian support at EL0. The SCTLR_EL1.E0E bit can be configured.

All other values are reserved.

This field is invalid and is RES0 if ID_AA64MMFR0_EL1.BigEnd is not 0b0000.

SNSMem, bits [15:12]

Indicates support for a distinction between Secure and Non-secure Memory. Defined values are:

0b0000   Does not support a distinction between Secure and Non-secure Memory.

0b0001   Does support a distinction between Secure and Non-secure Memory.

———— **Note** ————

If EL3 is implemented, the value 0b0000 is not permitted.

All other values are reserved.

BigEnd, bits [11:8]

Indicates support for mixed-endian configuration. Defined values are:

0b0000   No mixed-endian support. The SCTLR_ELx.EE bits have a fixed value. See the
           BigEndEL0 field, bits[19:16], for whether EL0 supports mixed-endian.

0b0001   Mixed-endian support. The SCTLR_ELx.EE and SCTLR_EL1.E0E bits can be
           configured.

All other values are reserved.

ASIDBits, bits [7:4]

Number of ASID bits. Defined values are:

0b0000   8 bits.

0b0010   16 bits.

All other values are reserved.

PARange, bits [3:0]

Physical Address range supported. Defined values are:

0b0000	32 bits, 4GB.
0b0001	36 bits, 64GB.
0b0010	40 bits, 1TB.
0b0011	42 bits, 4TB.
0b0100	44 bits, 16TB.
0b0101	48 bits, 256TB.
0b0110	52 bits, 4PB.

All other values are reserved.

The value 0b0110 is permitted only if the implementation includes FEAT_LPA, otherwise it is reserved.

Accessing the ID_AA64MMFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64MMFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64MMFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64MMFR0_EL1;
```

G1.3.8 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 state.
>
> For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

Configurations

> There are no configuration notes.

Attributes

> ID_AA64MMFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR1_EL1 bit assignments are:

63	32	31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0		XNX		SpecSEI		PAN		RES0		HPDS		RES0		VMIDBits		HAFDBS	

Bits [63:32]

> Reserved, RES0.

XNX, bits [31:28]

> Indicates support for execute-never control distinction by Exception level at stage 2. Defined values are:
>
> 0b0000   Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
>
> 0b0001   Distinction between EL0 and EL1 execute-never control at stage 2 supported.
>
> All other values are reserved.
>
> FEAT_XNX implements the functionality identified by the value 0b0001.
>
> From Armv8.2, the only permitted value is 0b0001.

SpecSEI, bits [27:24]

> Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches. The defined values of this field are:
>
> 0b0000   The PE never generates an SError interrupt due to an External abort on a speculative read.
>
> 0b0001   The PE might generate an SError interrupt due to an External abort on a speculative read.
>
> All other values are reserved.

PAN, bits [23:20]

> Privileged Access Never. Indicates support for the PAN bit in PSTATE, SPSR_EL1, SPSR_EL2, SPSR_EL3, and DSPSR_EL0. Defined values are:
>
> 0b0000   PAN not supported.
>
> 0b0001   PAN supported.
>
> 0b0010   PAN supported and AT S1E1RP and AT S1E1WP instructions supported.
>
> All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

In Armv8.1, the permitted values are 0b0001 and 0b0011.

From Armv8.2, the permitted values are 0b0010 and 0b0011.

Bits [19:16]

Reserved, RES0.

HPDS, bits [15:12]

Hierarchical Permission Disables. Indicates support for disabling hierarchical controls in translation tables. Defined values are:

0b0000   Disabling of hierarchical controls not supported.

0b0001   Disabling of hierarchical controls supported with the TCR_EL1.{HPD1, HPD0}, TCR_EL2.HPD or TCR_EL2.{HPD1, HPD0}, and TCR_EL3.HPD bits.

0b0010   As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

FEAT_HPDS implements the functionality identified by the value 0b0001.

FEAT_HPDS2 implements the functionality identified by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

Bits [11:8]

Reserved, RES0.

VMIDBits, bits [7:4]

Number of VMID bits. Defined values are:

0b0000   8 bits

0b0010   16 bits

All other values are reserved.

FEAT_VMID16 implements the functionality identified by the value 0b0010.

From Armv8.1, the permitted values are 0b0000 and 0b0010.

HAFDBS, bits [3:0]

Hardware updates to Access flag and Dirty state in translation tables. Defined values are:

0b0000   Hardware update of the Access flag and dirty state are not supported.

0b0001   Hardware update of the Access flag is supported.

0b0010   Hardware update of both the Access flag and dirty state is supported.

All other values are reserved.

FEAT_HAFDBS implements the functionality identified by the values 0b0001 and 0b0010.

From Armv8.1, the permitted values are 0b0000, 0b0001, and 0b0010.

Accessing the ID_AA64MMFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64MMFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64MMFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64MMFR1_EL1;
```

G1.3.9 ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2

The ID_AA64MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.
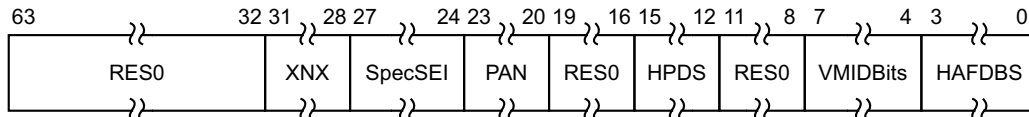
Configurations

—————— **Note** ——————

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_AA64MMFR2_EL1 is a 64-bit register.

Field descriptions

The ID_AA64MMFR2_EL1 bit assignments are:

Bits [63:56]

Reserved, RES0.

BBM, bits [55:52]

Allows identification of the requirements of the hardware to have break-before-make sequences when changing block size for a translation.

0b0000   Level 0 support for changing block size is supported.

0b0001   Level 1 support for changing block size is supported.

0b0010   Level 2 support for changing block size is supported.

All other values are reserved.

FEAT_BBM implements the functionality identified by the values 0b0000, 0b0001, and 0b0010.

From Armv8.4, the permitted values are 0b0000, 0b0001, and 0b0010.

TTL, bits [51:48]

Indicates support for TTL field in address operations. Defined values are:

0b0000   TLB maintenance instructions by address have bits[47:44] as RES0.

0b0001   TLB maintenance instructions by address have bits[47:44] holding the TTL field.

All other values are reserved.

FEAT_TTL implements the functionality identified by the value 0b0001.

This field affects TLBI IPAS2E1, TLBI IPAS2E1IS, TLBI IPAS2E1OS, TLBI IPAS2LE1, TLBI IPAS2LE1IS, TLBI IPAS2LE1OS, TLBI VAAE1, TLBI VAAE1IS, TLBI VAAE1OS, TLBI VAALE1, TLBI VAALE1IS, TLBI VAALE1OS, TLBI VAE1, TLBI VAE1IS, TLBI VAE1OS, TLBI VAE2, TLBI VAE2IS, TLBI VAE2OS, TLBI VAE3, TLBI VAE3IS, TLBI VAE3OS, TLBI VALE1, TLBI VALE1IS, TLBI VALE1OS, TLBI VALE2, TLBI VALE2IS, TLBI VALE2OS, TLBI VALE3, TLBI VALE3IS, TLBI VALE3OS.

From Armv8.4, the only permitted value is 0b0001.

Bits [47:44]

Reserved, RES0.

FWB, bits [43:40]

Indicates support for HCR_EL2.FWB. Defined values are:

0b0000   HCR_EL2.FWB bit is not supported.

0b0001   HCR_EL2.FWB is supported.

All other values reserved.

FEAT_S2FWB implements the functionality identified by the value 0b0001.

From Armv8.4, the only permitted value is 0b0001.

IDS, bits [39:36]

Indicates the value of ESR_ELx.EC that reports an exception generated by a read access to the feature ID space. Defined values are:

0b0000   An exception which is generated by a read access to the feature ID space, other than a trap caused by HCR_EL2.TIDx, SCTLR_EL1.UCT, or SCTLR_EL2.UCT, is reported by ESR_ELx.EC == 0x0.

0b0001   All exceptions generated by an AArch64 read access to the feature ID space are reported by ESR_ELx.EC == 0x18.

All other values are reserved.

The Feature ID space is defined as the System register space in AArch64 with op0==3, op1=={0, 1, 3}, CRn==0, CRm=={0-7}, op2=={0-7}.

FEAT_IDST implements the functionality identified by the value 0b0001.

From Armv8.4, the only permitted value is 0b0001.

AT, bits [35:32]

Identifies support for unaligned single-copy atomicity and atomic functions. Defined values are:

0b0000   Unaligned single-copy atomicity and atomic functions are not supported.

0b0001   Unaligned single-copy atomicity and atomic functions with a 16-byte address range aligned to 16-bytes are supported.

All other values are reserved.

FEAT_LSE2 implements the functionality identified by the value 0b0001.

In Armv8.2, the permitted values are 0b0000 and 0b0001.

From Armv8.4, the only permitted value is 0b0001.

ST, bits [31:28]

Identifies support for small translation tables. Defined values are:

0b0000   The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 39.

0b0001   The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 48 for 4KB and 16KB granules, and 47 for 64KB granules.

All other values are reserved.

FEAT_TTST implements the functionality identified by the value 0b0001.

If FEAT_SEL2 is implemented, the only permitted value is 0b0001.

In an implementation which does not support FEAT_SEL2, the permitted values are 0b0000 and 0b0001.

Bits [27:24]

Reserved, RES0.

CCIDX, bits [23:20]

> Support for the use of revised CCSIDR_EL1 register format. Defined values are:
>
> 0b0000   32-bit format implemented for all levels of the CCSIDR_EL1.
>
> 0b0001   64-bit format implemented for all levels of the CCSIDR_EL1.
>
> All other values are reserved.
>
> FEAT_CCIDX implements the functionality identified by the value 0b0001.
>
> From Armv8.3, the permitted values are 0b0000 and 0b0001.

VARange, bits [19:16]

> Indicates support for a larger virtual address. Defined values are:
>
> 0b0000   VMSAv8-64 supports 48-bit VAs.
>
> 0b0001   VMSAv8-64 supports 52-bit VAs when using the 64KB translation granule. The other translation granules support 48-bit VAs.
>
> All other values are reserved.
>
> FEAT_LVA implements the functionality identified by the value 0b0001.
>
> From Armv8.2, the permitted values are 0b0000 and 0b0001.

IESB, bits [15:12]

> Indicates support for the IESB bit in the SCTLR_ELx registers. Defined values are:
>
> 0b0000   IESB bit in the SCTLR_ELx registers is not supported.
>
> 0b0001   IESB bit in the SCTLR_ELx registers is supported.
>
> All other values are reserved.
>
> FEAT_IESB implements the functionality identified by the value 0b0001.

Bits [11:8]

> Reserved, RES0.

UAO, bits [7:4]

> User Access Override. Defined values are:
>
> 0b0000   UAO not supported.
>
> 0b0001   UAO supported.
>
> All other values are reserved.
>
> FEAT_UAO implements the functionality identified by the value 0b0001.
>
> From Armv8.2, the only permitted value is 0b0001.

CnP, bits [3:0]

> Indicates support for Common not Private translations. Defined values are:
>
> 0b0000   Common not Private translations not supported.
>
> 0b0001   Common not Private translations supported.
>
> All other values are reserved.
>
> FEAT_TTCNP implements the functionality identified by the value 0b0001.
>
> From Armv8.2, the only permitted value is 0b0001.

Accessing the ID_AA64MMFR2_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64MMFR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64MMFR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64MMFR2 trapped by
HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64MMFR2_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64MMFR2_EL1;
```

G1.3.10 ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0

The ID_AA64PFR0_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

Configurations

The external register EDPFR gives information from this register.

Attributes

ID_AA64PFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR0_EL1 bit assignments are:
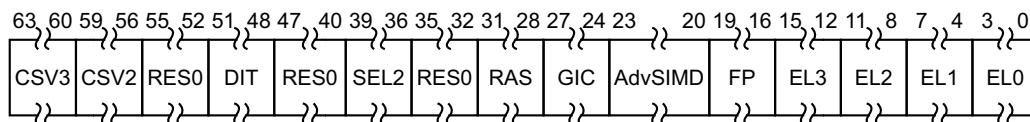
CSV3, bits [63:60]

Speculative use of faulting data. Defined values are:

0b0000   This Device does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

0b0001   Data loaded under speculation with a permission or domain fault cannot be used to form an address or generate condition codes or SVE predicate values to be used by instructions newer than the load in the speculative sequence

All other values are reserved.

FEAT_CSV3 implements the functionality identified by the value 0b0001.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

In Armv8-R, the only permitted value is 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

CSV2, bits [59:56]

Speculative use of out of context branch targets. Defined values are:

0b0000   This Device does not disclose whether branch targets trained in one hardware described context can affect speculative execution in a different hardware described context.

0b0001   Branch targets trained in one hardware described context can only affect speculative execution in a different hardware described context in a hard-to-determine way. Contexts do not include the SCXTNUM_ELx register contexts, and these registers are not supported.

0b0010   Branch targets trained in one hardware described context can only affect speculative execution in a different hardware described context in a hard-to-determine way. Contexts include the SCXTNUM_ELx register contexts, and these registers are supported.

In Armv8-R, the only permitted values are 0b0001 or 0b0010.

All other values are reserved.

Bits [55:52]

    Reserved, RES0.

DIT, bits [51:48]

    Data Independent Timing. Defined values are:

    0b0000   AArch64 does not guarantee constant execution time of any instructions.

    0b0001   AArch64 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

    All other values are reserved.

    FEAT_DIT implements the functionality identified by the value 0b0001.

    From Armv8.4, the only permitted value is 0b0001.

Bits [47:40]

    Reserved, RES0.

SEL2, bits [39:36]

    Secure EL2. Defined values are:

    0b0000   Secure EL2 is not implemented.

    0b0001   Secure EL2 is implemented.

    All other values are reserved.

    FEAT_SEL2 implements the functionality identified by the value 0b0001.

Bits [35:32]

    Reserved, RES0.

RAS, bits [31:28]

    RAS Extension version. Defined values are:

    0b0000   No RAS Extension.

    0b0001   RAS Extension present.

    0b0010   FEAT_RASv1p1 present. As 0b0001, and adds support for:

        •   Additional ERXMISC<m>_EL1 System registers.

        •   Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension.

        Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

    All other values are reserved.

    FEAT_RASv1p1 and FEAT_DoubleFault implement the functionality identified by the value 0b0010.

    In Armv8.0 and Armv8.1, the permitted values are 0b0000 and 0b0001.

    In Armv8.2, the only permitted value is 0b0001.

    From Armv8.4, when FEAT_DoubleFault is not implemented, and ERRIDR_EL1.NUM is zero, the permitted values are IMPLEMENTATION DEFINED 0b0001 or 0b0010. Otherwise from Armv8.4, the only permitted value is 0b0010.

GIC, bits [27:24]

    System register GIC CPU interface. Defined values are:

    0b0000   GIC CPU interface system registers not implemented.

    0b0001   System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.

0b0011   System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

AdvSIMD, bits [23:20]

Advanced SIMD. Defined values are:

0b0000   Advanced SIMD is implemented, including support for the following SISD and SIMD operations:

- Integer byte, halfword, word and doubleword element operations.

- Single-precision and double-precision floating-point arithmetic.

- Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

0b0001   As for 0b0000, and also includes support for half-precision floating-point arithmetic.

0b1111   Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- 0b0000 in an implementation with Advanced SIMD support that does not include the FEAT_FP16 extension.

- 0b0001 in an implementation with Advanced SIMD support that includes the FEAT_FP16 extension.

- 0b1111 in an implementation without Advanced SIMD support.

FP, bits [19:16]

Floating-point. Defined values are:

0b0000   Floating-point is implemented, and includes support for:

- Single-precision and double-precision floating-point types.

- Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.

0b0001   As for 0b0000, and also includes support for half-precision floating-point arithmetic.

0b1111   Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- 0b0000 in an implementation with floating-point support that does not include the FEAT_FP16 extension.

- 0b0001 in an implementation with floating-point support that includes the FEAT_FP16 extension.

- 0b1111 in an implementation without floating-point support.

EL3, bits [15:12]

EL3 Exception level handling. In Armv8-R, the only permitted value is 0b0000. All other values are reserved.

0b0000   EL3 is not implemented.

EL2, bits [11:8]

EL2 Exception level handling. In Armv8-R, the only permitted value is 0b0001. All other values are reserved.

0b0001   EL2 can be executed in AArch64 state only.

EL1, bits [7:4]

> EL1 Exception level handling. In Armv8-R, the only permitted value is `0b0001`. All other values are reserved.
>
> `0b0001`   EL1 can be executed in AArch64 state only.

EL0, bits [3:0]

> EL0 Exception level handling. In Armv8-R, the only permitted value is `0b0001`. All other values are reserved.
>
> `0b0001`   EL0 can be executed in AArch64 state only.

Accessing the ID_AA64PFR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64PFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64PFR0_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64PFR0_EL1;
```

G1.3.11   ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers.

Configurations

There are no configuration notes.

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR1_EL1 bit assignments are:
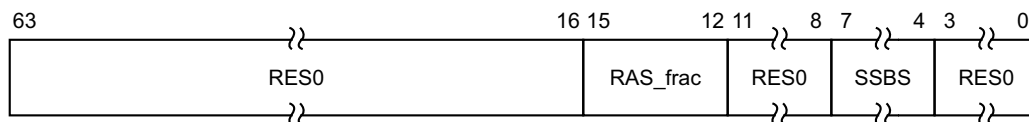
Bits [63:16]

Reserved, RES0.

RAS_frac, bits [15:12]

RAS Extension fractional field. Defined values are:

0b0000   If ID_AA64PFR0_EL1.RAS == 0b0001, RAS Extension implemented.

0b0001   If ID_AA64PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for:

- Additional ERXMISC<m>_EL1 System registers.

- Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS, and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

FEAT_RASv1p1 implements the functionality identified by the value 0b0001.

This field is valid only if ID_AA64PFR0_EL1.RAS == 0b0001.

Bits [11:8]

Reserved, RES0.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:

0b0000   AArch64 provides no mechanism to control the use of Speculative Store Bypassing.

0b0001   AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

0b0010   AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypassing Safe, and the MSR and MRS instructions to directly read and write the PSTATE.SSBS field.

All other values are reserved.

FEAT_SSBS implements the functionality identified by the value `0b0001`.

FEAT_SSBS implements the functionality identified by the value `0b0010`.

Bits [3:0]

Reserved, RES0.

Accessing the ID_AA64PFR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, ID_AA64PFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64PFR1_EL1;
elsif PSTATE.EL == EL2 then
    return ID_AA64PFR1_EL1;
```

G1.3.12   MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

Purpose

If VMSAv8-64 is enabled at stage 1 of EL1&0 translation regime, this register provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

If PMSAv8-64 is enabled at stage 1 of EL1&0 translation regime, this register provides the memory attribute encodings corresponding to the possible AttrIndx values in PRLAR_EL1 register for stage 1 translations.

Configurations

There are no configuration notes.

Attributes

MAIR_EL1 is a 64-bit register.

Field descriptions

The MAIR_EL1 bit assignments are:
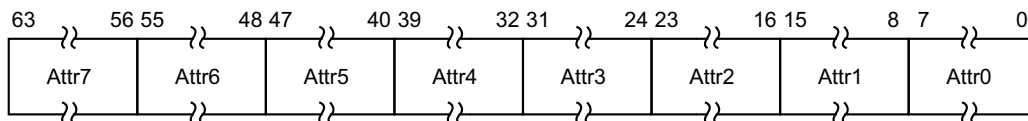
MAIR_EL1 is permitted to be cached in a TLB.

Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr	Meaning
0b0000dd00	Device memory. See encoding of 'dd' for the type of Device memory.
0b0000ddxx, (xx != 00)	UNPREDICTABLE
0boooiiii, (oooo != 0000 and iiii != 0000)	Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal Memory.
0b11110000	If FEAT_MTE is implemented, then: Tagged Normal Memory. Inner+Outer Write-Back Non-transient memory, Inner+Outer Read-Allocate, Inner+Outer Write-Allocate. Otherwise, UNPREDICTABLE.
0bxxxx0000, (xxxx != 0000 and xxxx != 1111)	UNPREDICTABLE

'dd' is encoded as follows:

dd	Meaning
0b00	Device-nGnRnE memory
0b01	Device-nGnRE memory
0b10	Device-nGRE memory
0b11	Device-GRE memory

'oooo' is encoded as follows:

'oooo'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not0b00	Normal memory, Outer Write-Through Transient
0b0100	Normal memory, Outer Non-cacheable
0b01RW, RW not0b00	Normal memory, Outer Write-Back Transient
0b10RW	Normal memory, Outer Write-Through Non-transient
0b11RW	Normal memory, Outer Write-Back Non-transient

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iiii' is encoded as follows:

'iiii'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not0b00	Normal memory, Inner Write-Through Transient
0b0100	Normal memory, Inner Non-cacheable
0b01RW, RW not0b00	Normal memory, Inner Write-Back Transient
0b10RW	Normal memory, Inner Write-Through Non-transient
0b11RW	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'oooo' and 'iiii' fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the MAIR_EL1

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.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return MAIR_EL1;
elsif PSTATE.EL == EL2 then
    return MAIR_EL1;
```

MSR MAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MAIR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    MAIR_EL1 = X[t];
```

G1.3.13 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in PRLAR_EL2 for stage 1 EL2 translation regime and for stage 2 EL1&0 translation regime.

For stage 2 EL1&0 translations, the memory attributes are derived from MAIR_EL2 register as described in the Armv8-R AArch64 architecture.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

MAIR_EL2 is a 64-bit register.

Field descriptions

The MAIR_EL2 bit assignments are:
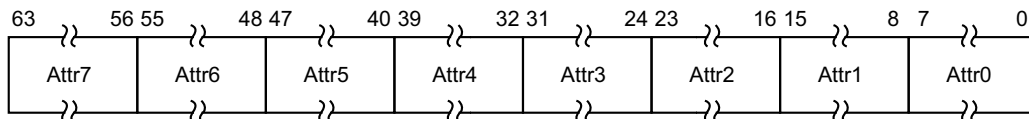
MAIR_EL2 is permitted to be cached in a TLB.

Attr<n>, bits [8n+7:8n], for n = 0 to 7

The memory attribute encoding for an AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr	Meaning
0b0000dd00	Device memory. See encoding of 'dd' for the type of Device memory.
0b0000ddxx, (xx != 00)	UNPREDICTABLE
0booooiiii, (oooo != 0000 and iiii != 0000)	Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal Memory.
0b11110000	If FEAT_MTE is implemented, then: Tagged Normal Memory. Inner+Outer Write-Back Non-transient memory, Inner+Outer Read-Allocate, Inner+Outer Write-Allocate. Otherwise, UNPREDICTABLE.
0bxxxx0000, (xxxx != 0000 and xxxx != 1111)	UNPREDICTABLE

'dd' is encoded as follows:

dd	Meaning
0b00	Device-nGnRnE memory
0b01	Device-nGnRE memory
0b10	Device-nGRE memory
0b11	Device-GRE memory

'oooo' is encoded as follows:

'oooo'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not0b00	Normal memory, Outer Write-Through Transient
0b0100	Normal memory, Outer Non-cacheable
0b01RW, RW not0b00	Normal memory, Outer Write-Back Transient
0b10RW	Normal memory, Outer Write-Through Non-transient
0b11RW	Normal memory, Outer Write-Back Non-transient

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iiii' is encoded as follows:

'iiii'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not0b00	Normal memory, Inner Write-Through Transient
0b0100	Normal memory, Inner Non-cacheable
0b01RW, RW not0b00	Normal memory, Inner Write-Back Transient
0b10RW	Normal memory, Inner Write-Through Non-transient
0b11RW	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'oooo' and 'iiii' fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

This field resets to an architecturally UNKNOWN value.

Accessing the MAIR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic MAIR_EL2 or MAIR_EL1 is not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MAIR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return MAIR_EL2;
```

MSR MAIR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    MAIR_EL2 = X[t];
```

MRS <Xt>, MAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return MAIR_EL1;
elsif PSTATE.EL == EL2 then
    return MAIR_EL1;
```

MSR MAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MAIR_EL1 = X[t];
```

```
elsif PSTATE.EL == EL2 then
    MAIR_EL1 = X[t];
```

G1.3.14 MPUIR_EL1, MPU Type Register (EL1)

The MPUIR_EL1 characteristics are:

Purpose

Identifies the number of regions supported by the EL1 MPU.

Configurations

There are no configuration notes.

Attributes

MPUIR_EL1 is a 64-bit register.

Field descriptions

The MPUIR_EL1 bit assignments are:
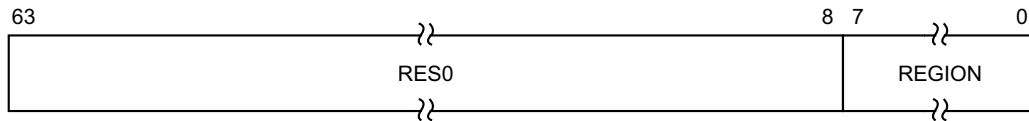
Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of EL1 MPU regions supported.

Accessing the MPUIR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MPUIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("FEAT_IDST") then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return MPUIR_EL1;
elsif PSTATE.EL == EL2 then
    return MPUIR_EL1;
```

G1.3.15 MPUIR_EL2, MPU Type Register (EL2)

The MPUIR_EL2 characteristics are:

Purpose

Identifies the number of regions supported by the EL2 MPU.

Configurations

There are no configuration notes.

Attributes

MPUIR_EL2 is a 64-bit register.

Field descriptions

The MPUIR_EL2 bit assignments are:

63	8 7	0
RES0		REGION
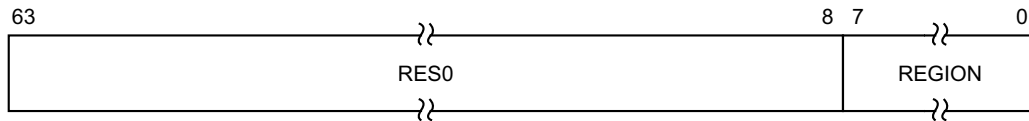
Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of EL2 MPU regions supported.

Accessing the MPUIR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MPUIR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return MPUIR_EL2;
```

G1.3.16 PRBAR_EL1, Protection Region Base Address Register (EL1)

The PRBAR_EL1 characteristics are:

Purpose

> Provides access to the base addresses for the EL1 MPU region. PRSELR_EL1.REGION determines which MPU region is selected.

Configurations

> All bits above implemented physical address range in this register should be treated as RES0.

Attributes

> PRBAR_EL1 is a 64-bit register.

Field descriptions

The PRBAR_EL1 bit assignments are:

Bits [63:52]

> Reserved, RES0.

BASE[51:48], bits [51:48]

> ***When FEAT_LPA is implemented:***
>
> > Extension to BASE[47:6]. When FEAT_LPA is implemented, BASE[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.
> >
> > This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
> > Reserved, RES0.

BASE[47:6], bits [47:6]

> Bits[47:6] of the lower inclusive limit of the selected EL1 MPU memory region. This value is zero extended to provide the base address to be checked against.
>
> This field resets to an architecturally UNKNOWN value.

SH[1:0], bits [5:4]

> Shareability attribute.
>
> | | |
> |---|---|
> | 0b00 | Non-shareable |
> | 0b01 | Reserved, CONSTRAINED UNPREDICTABLE |
> | 0b10 | Outer Shareable |
> | 0b11 | Inner Shareable |
>
> This field resets to an architecturally UNKNOWN value.

AP[2:1], bits [3:2]

> Access Permission attributes.
>
> | | |
> |---|---|
> | 0b00 | Read/write at EL1, no access at EL0 |
> | 0b01 | Read/write at EL1 and EL0 |
> | 0b10 | Read-only at EL1, no access at EL0 |
> | 0b11 | Read-only at EL1 and EL0 |
>
> This field resets to an architecturally UNKNOWN value.

XN, bit [1]

> Execute Never
>
> | | |
> |---|---|
> | 0b0 | Execution of instructions fetched from the region is permitted. |
> | 0b1 | Execution of instructions fetched from the region is not permitted. |
>
> This field resets to an architecturally UNKNOWN value.

Bit [0]

> Reserved, RES0.

Accessing the PRBAR_EL1

Any access to MPU region register PRBAR_EL1 above the number of implemented regions specified by MPUIR_EL1.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR_EL1 register return an UNKNOWN value.

- Writes to unimplemented PRBAR_EL1 register make all PRBAR_EL1 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRBAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return PRBAR_EL1;
elsif PSTATE.EL == EL2 then
    return PRBAR_EL1;
```

MSR PRBAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRBAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    PRBAR_EL1 = X[t];
```

G1.3.17   PRBAR_EL2, Protection Region Base Address Register (EL2)

The PRBAR_EL2 characteristics are:

Purpose

> Provides access to the base addresses for the EL2 MPU region. PRSELR_EL2.REGION determines which MPU region is selected.

Configurations

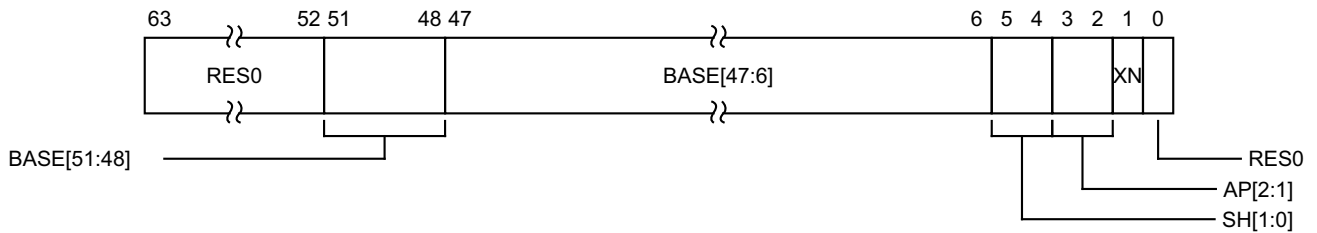> All bits above implemented physical address range in this register should be treated as RES0.

Attributes

> PRBAR_EL2 is a 64-bit register.

Field descriptions

The PRBAR_EL2 bit assignments are:

Bits [63:52]

> Reserved, RES0.

BASE[51:48], bits [51:48]

> ***When FEAT_LPA is implemented:***

> > Extension to BASE[47:6]. When FEAT_LPA is implemented, BASE[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

> > This field resets to an architecturally UNKNOWN value.

> ***Otherwise:***

> > Reserved, RES0.

BASE[47:6], bits [47:6]

> Bits[47:6] of the lower inclusive limit of the selected EL2 MPU memory region. This value is zero extended to provide the base address to be checked against.

> This field resets to an architecturally UNKNOWN value.

SH[1:0], bits [5:4]

> Shareability attribute.

> | | |
> |---|---|
> | 0b00 | Non-shareable |
> | 0b01 | Reserved, CONSTRAINED UNPREDICTABLE |
> | 0b10 | Outer Shareable |
> | 0b11 | Inner Shareable |

> This field resets to an architecturally UNKNOWN value.

AP[2:1], bits [3:2]

Access Permission attributes.

0b00   Read/write at EL2, no access at EL1 or EL0

0b01   Read/write at EL2, EL1 and EL0

0b10   Read-only at EL2, no access at EL1 or EL0

0b11   Read-only at EL2, EL1 and EL0

This field resets to an architecturally UNKNOWN value.

XN, bits [1:0]

Execute Never. For

- Stage 1 EL2 translation regime and

- Stage 2 EL1&0 translation regime when FEAT_XNX is not implemented

XN[1] determines whether execution of the instructions fetched from the MPU memory region is permitted. In this case, XN[0] is RES0

For stage 2 EL1&0 translation regime when FEAT_XNX is implemented, the behavior of XN[1:0] is same as that defined by VMSAv8-64 for EL1&0 stage 2 translation table XN[1:0],bits[54:53] field in Armv8-A architecture.

0b00   Execution of instructions fetched from the region is permitted.

0b01   Execution of instructions fetched from the region is not permitted.

This field resets to an architecturally UNKNOWN value.

Accessing the PRBAR_EL2

Any access to MPU region register PRBAR_EL2 above the number of implemented regions specified by MPUIR_EL2.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR_EL2 register return an UNKNOWN value.

- Writes to unimplemented PRBAR_EL2 register make all PRBAR_EL2 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRBAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return PRBAR_EL2;
```

MSR PRBAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRBAR_EL2 = X[t];
```

G1.3.18 PRBAR<n>_EL1, Protection Region Base Address Register n (EL1), n = 1 - 15

The PRBAR<n>_EL1 characteristics are:

Purpose

> Provides access to the base address for the MPU region determined by the value of 'n' and
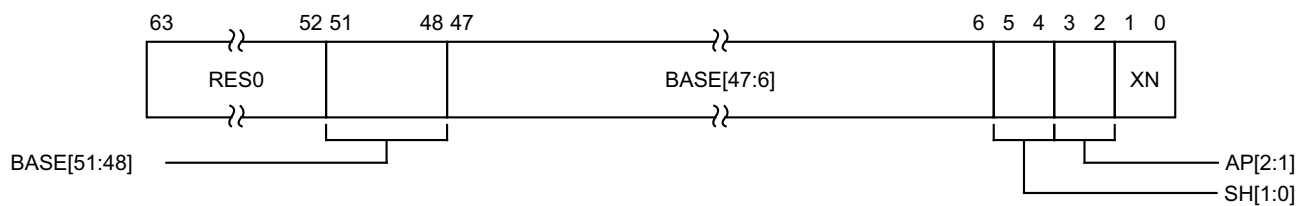> PRSELR_EL1.REGION as PRSELR_EL1.REGION<7:4>:n.

Configurations

> All bits above implemented physical address range in this register should be treated as RES0.

Attributes

> PRBAR<n>_EL1 is a 64-bit register.

Field descriptions

The PRBAR<n>_EL1 bit assignments are:

Bits [63:52]

> Reserved, RES0.

BASE[51:48], bits [51:48]

> ***When FEAT_LPA is implemented:***
>
> > Extension to BASE[47:6]. When FEAT_LPA is implemented, BASE[51:48] form the upper part of
> > the address value. Otherwise, for implementations with fewer than 52 physical address bits, the
> > upper bits of this field, corresponding to address bits that are not implemented, are RES0.
> >
> > This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
> > Reserved, RES0.

BASE[47:6], bits [47:6]

> Bits[47:6] of the lower inclusive limit of the selected EL1 MPU memory region. This value is zero
> extended to provide the base address to be checked against.
>
> This field resets to an architecturally UNKNOWN value.

SH[1:0], bits [5:4]

> Shareability attribute.
>
> | | |
> |---|---|
> | 0b00 | Non-shareable |
> | 0b01 | Reserved, CONSTRAINED UNPREDICTABLE |
> | 0b10 | Outer Shareable |
> | 0b11 | Inner Shareable |
>
> This field resets to an architecturally UNKNOWN value.

AP[2:1], bits [3:2]

> Access Permission attributes.
>
> 0b00   Read/write at EL1, no access at EL0
>
> 0b01   Read/write at EL1 and EL0
>
> 0b10   Read-only at EL1, no access at EL0
>
> 0b11   Read-only at EL1 and EL0
>
> This field resets to an architecturally UNKNOWN value.

XN, bit [1]

> Execute Never
>
> 0b0   Execution of instructions fetched from the region is permitted.
>
> 0b1   Execution of instructions fetched from the region is not permitted.
>
> This field resets to an architecturally UNKNOWN value.

Bit [0]

> Reserved, RES0.

Accessing the PRBAR<n>_EL1

Any access to MPU region register PRBAR<n>_EL1 above the number of implemented regions specified by MPUIR_EL1.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

*   Reads of unimplemented PRBAR<n>_EL1 return an UNKNOWN value.

*   Writes to unimplemented PRBAR<n>_EL1 register make all PRBAR_EL1 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRBAR<n>_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1:n[3:1]	n[0]:0b00

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return PRBAR_EL1[UInt(CRm<2:0>:op2<2>)];
elsif PSTATE.EL == EL2 then
    return PRBAR_EL1[UInt(CRm<2:0>:op2<2>)];
```

MSR PRBAR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1:n[3:1]	n[0]:0b00

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRBAR_EL1[UInt(CRm<2:0>:op2<2>)] = X[t];
elsif PSTATE.EL == EL2 then
    PRBAR_EL1[UInt(CRm<2:0>:op2<2>)] = X[t];
```

G1.3.19   PRBAR<n>_EL2, Protection Region Base Address Register n (EL2), n = 1 - 15

The PRBAR<n>_EL2 characteristics are:

Purpose

Provides access to the base address for the MPU region determined by the value of 'n' and
PRSELR_EL2.REGION as PRSELR_EL2.REGION<7:4>:n.
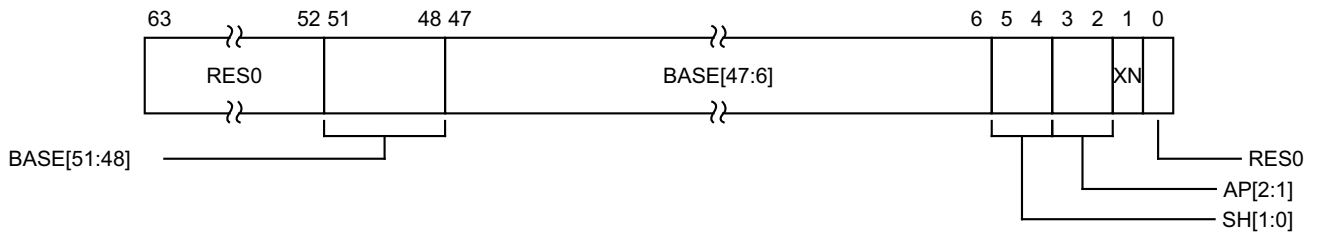
Configurations

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRBAR<n>_EL2 is a 64-bit register.

Field descriptions

The PRBAR<n>_EL2 bit assignments are:

Bits [63:52]

Reserved, RES0.

BASE[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to BASE[47:6]. When FEAT_LPA is implemented, BASE[51:48] form the upper part of
the address value. Otherwise, for implementations with fewer than 52 physical address bits, the
upper bits of this field, corresponding to address bits that are not implemented, are RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BASE[47:6], bits [47:6]

Bits[47:6] of the lower inclusive limit of the selected EL2 MPU memory region. This value is zero
extended to provide the base address to be checked against.

This field resets to an architecturally UNKNOWN value.

SH[1:0], bits [5:4]

Shareability attribute.

0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

This field resets to an architecturally UNKNOWN value.

AP[2:1], bits [3:2]

Access Permission attributes.

0b00	Read/write at EL2, no access at EL1 or EL0
0b01	Read/write at EL2, EL1 and EL0
0b10	Read-only at EL2, no access at EL1 or EL0
0b11	Read-only at EL2, EL1 and EL0

This field resets to an architecturally UNKNOWN value.

XN, bits [1:0]

Execute Never. For

- Stage 1 EL2 translation regime and

- Stage 2 EL1&0 translation regime when FEAT_XNX is not implemented

XN[1] determines whether execution of the instructions fetched from the MPU memory region is permitted. In this case, XN[0] is RES0

For stage 2 EL1&0 translation regime when FEAT_XNX is implemented, the behavior of XN[1:0] is same as that defined by VMSAv8-64 for EL1&0 stage 2 translation table XN[1:0],bits[54:53] field in Armv8-A architecture.

0b00	Execution of instructions fetched from the region is permitted.
0b01	Execution of instructions fetched from the region is not permitted.

This field resets to an architecturally UNKNOWN value.

Accessing the PRBAR<n>_EL2

Any access to MPU region register PRBAR<n>_EL2 above the number of implemented regions specified by MPUIR_EL2.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR<n>_EL2 return an UNKNOWN value.

- Writes to unimplemented PRBAR<n>_EL2 register make all PRBAR_EL2 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRBAR<n>_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1:n[3:1]	n[0]:0b00

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return PRBAR_EL2[UInt(CRm<2:0>:op2<2>)];
```

MSR PRBAR<n>_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1:n[3:1]	n[0]:0b00

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRBAR_EL2[UInt(CRm<2:0>:op2<2>)] = X[t];
```

G1.3.20 PRENR_EL1, Protection Region Enable Register (EL1)

The PRENR_EL1 characteristics are:

Purpose

Provides direct access to the PRLAR_EL1.EN bits of EL1 MPU regions from 0 to 31.

Configurations

There are no configuration notes.

Attributes

PRENR_EL1 is a 64-bit register.

Field descriptions

The PRENR_EL1 bit assignments are:
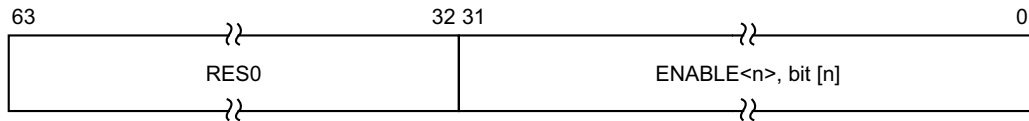
63	32	31	0
RES0		ENABLE<n>, bit [n]	

Bits [63:32]

Reserved, RES0.

ENABLE<n>, bit [n], for n = 0 to 31

Enable bit. Each bit, n, enables or disables the respective EL1 MPU region. The bits associated with the unimplemented MPU regions are RAZ/WI.

0b0   Disables the EL1 MPU n region.

0b1   Enables the EL1 MPU n region.

This field resets to 0.

Accessing the PRENR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRENR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return PRENR_EL1;
elsif PSTATE.EL == EL2 then
    return PRENR_EL1;
```

MSR PRENR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRENR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    PRENR_EL1 = X[t];
```

G1.3.21 PRENR_EL2, Protection Region Enable Register (EL2)

The PRENR_EL2 characteristics are:

Purpose

Provides direct access to the PRLAR_EL2.EN bits of EL2 MPU regions from 0 to 31.

Configurations

There are no configuration notes.

Attributes

PRENR_EL2 is a 64-bit register.

Field descriptions

The PRENR_EL2 bit assignments are:
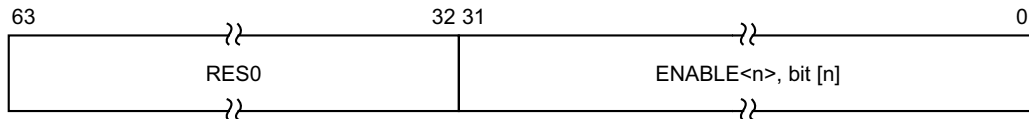
Bits [63:32]

Reserved, RES0.

ENABLE<n>, bit [n], for n = 0 to 31

Enable bit. Each bit, n, enables or disables the respective EL2 MPU region. The bits associated with the unimplemented MPU regions are RAZ/WI.

0b0   Disables the EL2 MPU n region.

0b1   Enables the EL2 MPU n region.

This field resets to 0.

Accessing the PRENR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRENR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return PRENR_EL2;
```

MSR PRENR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRENR_EL2 = X[t];
```

G1.3.22   PRLAR_EL1, Protection Region Limit Address Register (EL1)

The PRLAR_EL1 characteristics are:

Purpose

> Provides access to the limit addresses for the EL1 MPU region. PRSELR_EL1.REGION determines
> which MPU region is selected.

Configurations

> All bits above implemented physical address range in this register should be treated as RES0.

Attributes

> PRLAR_EL1 is a 64-bit register.

Field descriptions

The PRLAR_EL1 bit assignments are:

Bits [63:52]

> Reserved, RES0.

LIMIT[51:48], bits [51:48]

> *When FEAT_LPA is implemented:*
>
> Extension to LIMIT[47:6]. When FEAT_LPA is implemented, LIMIT[51:48] form the upper part
> of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the
> upper bits of this field, corresponding to address bits that are not implemented, are RES0.
>
> This field resets to an architecturally UNKNOWN value.
>
> *Otherwise:*
>
> Reserved, RES0.

LIMIT[47:6], bits [47:6]

> Bits[47:6] of the upper inclusive limit of the selected EL1 MPU memory region. This value is
> concatenated with the value 0x3F to provide the limit address to be checked against.
>
> This field resets to an architecturally UNKNOWN value.

Bit [5]

> Reserved, RES0.

NS, bit [4]

> Non-secure bit. Specifies whether the output address is in the Secure or Non-secure memory.
>
> 0b0   Output address is in Secure address space.
>
> 0b1   Output address is in Non-secure address space.
>
> This field resets to an architecturally UNKNOWN value.

AttrIndx[2:0], bits [3:1]

Selects attributes from within the associated Memory Attribute Indirection Register.

0b000   Select the Attr0 field from MAIR_EL1.

0b001   Select the Attr1 field from MAIR_EL1.

0b010   Select the Attr2 field from MAIR_EL1.

0b011   Select the Attr3 field from MAIR_EL1.

0b100   Select the Attr4 field from MAIR_EL1.

0b101   Select the Attr5 field from MAIR_EL1.

0b110   Select the Attr6 field from MAIR_EL1.

0b111   Select the Attr7 field from MAIR_EL1.

This field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

0b0   Region disabled.

0b1   Region enabled.

This field resets to 0.

Accessing the PRLAR_EL1

Any access to MPU region register PRLAR_EL1 above the number of implemented regions specified by MPUIR_EL1.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR_EL1 register return an UNKNOWN value.

- Writes to unimplemented PRLAR_EL1 register make all PRLAR_EL1 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRLAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return PRLAR_EL1;
elsif PSTATE.EL == EL2 then
    return PRLAR_EL1;
```

MSR PRLAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRLAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    PRLAR_EL1 = X[t];
```

G1.3.23   PRLAR_EL2, Protection Region Limit Address Register (EL2)

The PRLAR_EL2 characteristics are:

Purpose

Provides access to the limit addresses for the EL2 MPU region. PRSELR_EL2.REGION determines which MPU region is selected.
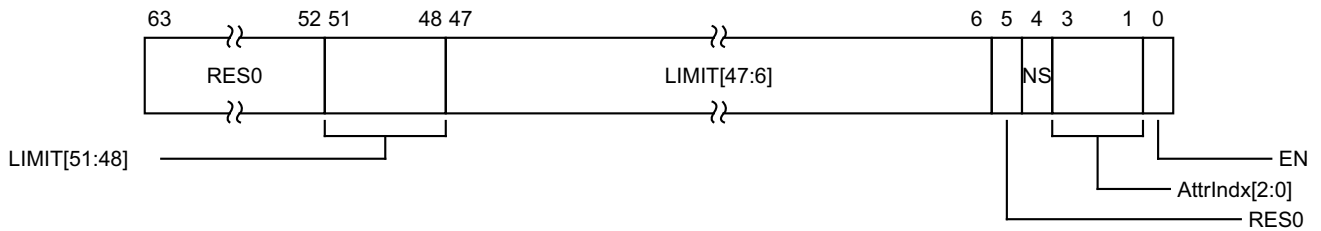
Configurations

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR_EL2 is a 64-bit register.

Field descriptions

The PRLAR_EL2 bit assignments are:

Bits [63:52]

Reserved, RES0.

LIMIT[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to LIMIT[47:6]. When FEAT_LPA is implemented, LIMIT[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LIMIT[47:6], bits [47:6]

Bits[47:6] of the upper inclusive limit of the selected EL2 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

NS, bit [4]

Non-secure bit. Specifies whether the output address is in the Secure or Non-secure memory.

0b0   Output address is in Secure address space.

0b1   Output address is in Non-secure address space.

This field resets to an architecturally UNKNOWN value.

AttrIndx[2:0], bits [3:1]

Selects attributes from within the associated Memory Attribute Indirection Register.

0b000　　Select the Attr0 field from MAIR_EL2.

0b001　　Select the Attr1 field from MAIR_EL2.

0b010　　Select the Attr2 field from MAIR_EL2.

0b011　　Select the Attr3 field from MAIR_EL2.

0b100　　Select the Attr4 field from MAIR_EL2.

0b101　　Select the Attr5 field from MAIR_EL2.

0b110　　Select the Attr6 field from MAIR_EL2.

0b111　　Select the Attr7 field from MAIR_EL2.

This field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

0b0　　Region disabled.

0b1　　Region enabled.

This field resets to 0.

Accessing the PRLAR_EL2

Any access to MPU region register PRLAR_EL2 above the number of implemented regions specified by MPUIR_EL2.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR_EL2 register return an UNKNOWN value.

- Writes to unimplemented PRLAR_EL2 register make all PRLAR_EL2 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRLAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return PRLAR_EL2;
```

MSR PRLAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```

```
        elsif PSTATE.EL == EL2 then
            PRLAR_EL2 = X[t];
```

G1.3.24 PRLAR<n>_EL1, Protection Region Limit Address Register n (EL1), n = 1 - 15

The PRLAR<n>_EL1 characteristics are:

Purpose

Provides access to the limit address for the MPU region determined by the value of 'n' and PRSELR_EL1.REGION as PRSELR_EL1.REGION<7:4>:n.
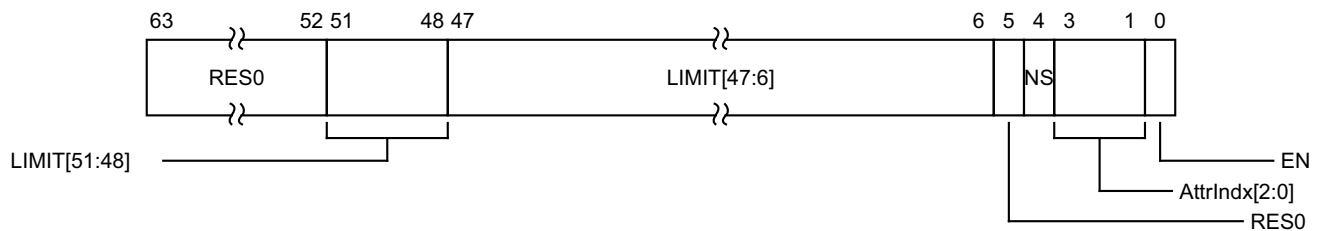
Configurations

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR<n>_EL1 is a 64-bit register.

Field descriptions

The PRLAR<n>_EL1 bit assignments are:

Bits [63:52]

Reserved, RES0.

LIMIT[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to LIMIT[47:6]. When FEAT_LPA is implemented, LIMIT[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LIMIT[47:6], bits [47:6]

Bits[47:6] of the upper inclusive limit of the selected EL1 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

NS, bit [4]

Non-secure bit. Specifies whether the output address is in the Secure or Non-secure memory.

0b0   Output address is in Secure address space.

0b1   Output address is in Non-secure address space.

This field resets to an architecturally UNKNOWN value.

AttrIndx[2:0], bits [3:1]

Selects attributes from within the associated Memory Attribute Indirection Register.

0b000	Select the Attr0 field from MAIR_EL1.
0b001	Select the Attr1 field from MAIR_EL1.
0b010	Select the Attr2 field from MAIR_EL1.
0b011	Select the Attr3 field from MAIR_EL1.
0b100	Select the Attr4 field from MAIR_EL1.
0b101	Select the Attr5 field from MAIR_EL1.
0b110	Select the Attr6 field from MAIR_EL1.
0b111	Select the Attr7 field from MAIR_EL1.

This field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

| 0b0 | Region disabled. |
| 0b1 | Region enabled. |

This field resets to 0.

Accessing the PRLAR<n>_EL1

Any access to MPU region register PRLAR<n>_EL1 above the number of implemented regions specified by MPUIR_EL1.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR<n>_EL1 return an UNKNOWN value.

- Writes to unimplemented PRLAR<n>_EL1 register make all PRLAR_EL1 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRLAR<n>_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1:n[3:1]	n[0]:0b01

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return PRLAR_EL1[UInt(CRm<2:0>:op2<2>)];
elsif PSTATE.EL == EL2 then
    return PRLAR_EL1[UInt(CRm<2:0>:op2<2>)];
```

MSR PRLAR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1:n[3:1]	n[0]:0b01

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRLAR_EL1[UInt(CRm<2:0>:op2<2>)] = X[t];
elsif PSTATE.EL == EL2 then
    PRLAR_EL1[UInt(CRm<2:0>:op2<2>)] = X[t];
```

G1.3.25 PRLAR<n>_EL2, Protection Region Limit Address Register n (EL2), n = 1 - 15

The PRLAR<n>_EL2 characteristics are:

Purpose

Provides access to the limit address for the MPU region determined by the value of 'n' and PRSELR_EL2.REGION as PRSELR_EL2.REGION<7:4>:n.
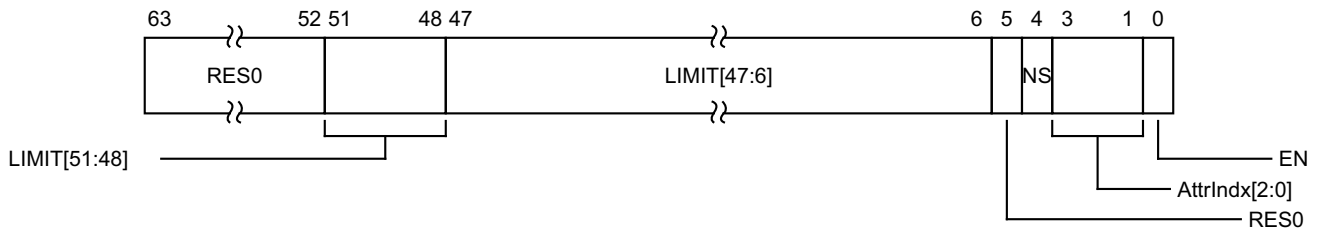
Configurations

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR<n>_EL2 is a 64-bit register.

Field descriptions

The PRLAR<n>_EL2 bit assignments are:

Bits [63:52]

Reserved, RES0.

LIMIT[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to LIMIT[47:6]. When FEAT_LPA is implemented, LIMIT[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LIMIT[47:6], bits [47:6]

Bits[47:6] of the upper inclusive limit of the selected EL2 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

NS, bit [4]

Non-secure bit. Specifies whether the output address is in the Secure or Non-secure memory.

0b0   Output address is in Secure address space.

0b1   Output address is in Non-secure address space.

This field resets to an architecturally UNKNOWN value.

AttrIndx[2:0], bits [3:1]

> Selects attributes from within the associated Memory Attribute Indirection Register.
>
> 0b000   Select the Attr0 field from MAIR_EL2.
>
> 0b001   Select the Attr1 field from MAIR_EL2.
>
> 0b010   Select the Attr2 field from MAIR_EL2.
>
> 0b011   Select the Attr3 field from MAIR_EL2.
>
> 0b100   Select the Attr4 field from MAIR_EL2.
>
> 0b101   Select the Attr5 field from MAIR_EL2.
>
> 0b110   Select the Attr6 field from MAIR_EL2.
>
> 0b111   Select the Attr7 field from MAIR_EL2.
>
> This field resets to an architecturally UNKNOWN value.

EN, bit [0]

> Region enable bit.
>
> 0b0   Region disabled.
>
> 0b1   Region enabled.
>
> This field resets to 0.

Accessing the PRLAR<n>_EL2

Any access to MPU region register PRLAR<n>_EL2 above the number of implemented regions specified by MPUIR_EL2.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR<n>_EL2 return an UNKNOWN value.

- Writes to unimplemented PRLAR<n>_EL2 register make all PRLAR_EL2 registers value UNKNOWN.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRLAR<n>_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1:n[3:1]	n[0]:0b01

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return PRLAR_EL2[UInt(CRm<2:0>:op2<2>)];
```

MSR PRLAR<n>_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1:n[3:1]	n[0]:0b01

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
```

```
elsif PSTATE.EL == EL2 then
    PRLAR_EL2[UInt(CRm<2:0>:op2<2>)] = X[t];
```

G1.3.26   PRSELR_EL1, Protection Region Selection Register (EL1)

The PRSELR_EL1 characteristics are:

Purpose

Selects the region number for the EL1 MPU region associated with the PRBAR_EL1 and PRLAR_EL1 registers.

Configurations

There are no configuration notes.

Attributes

PRSELR_EL1 is a 64-bit register.

Field descriptions

The PRSELR_EL1 bit assignments are:
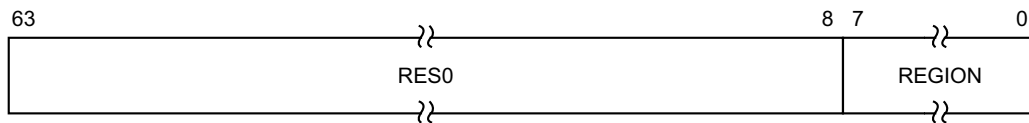
Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of the current EL1 MPU region visible in PRBAR_EL1 and PRLAR_EL1. For N implemented MPU regions, memory region numbering starts at 0 and increments by 1 to the value N-1.

Writing a value greater than or equal to the number of implemented MPU regions specified by MPUIR_EL1.REGION, results in CONSTRAINED UNPREDICTABLE behavior.

CONSTRAINED UNPREDICTABLE behavior is that PRSELR_EL1 register becomes UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the PRSELR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRSELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        return PRSELR_EL1;
elsif PSTATE.EL == EL2 then
    return PRSELR_EL1;
```

MSR PRSELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRSELR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    PRSELR_EL1 = X[t];
```

G1.3.27 PRSELR_EL2, Protection Region Selection Register (EL2)

The PRSELR_EL2 characteristics are:

Purpose

Selects the region number for the EL2 MPU region associated with the PRBAR_EL2 and PRLAR_EL2 registers.

Configurations

There are no configuration notes.

Attributes

PRSELR_EL2 is a 64-bit register.

Field descriptions

The PRSELR_EL2 bit assignments are:
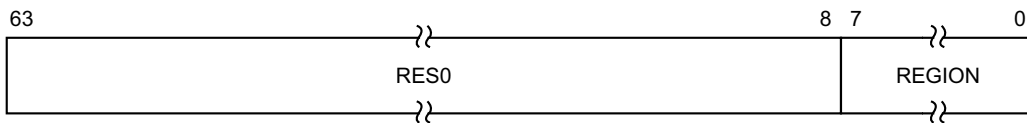
Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of the current EL2 MPU region visible in PRBAR_EL2 and PRLAR_EL2. For N implemented MPU regions, memory region numbering starts at 0 and increments by 1 to the value N-1.

Writing a value greater than or equal to the number of implemented MPU regions specified by MPUIR_EL2.REGION, results in CONSTRAINED UNPREDICTABLE behavior.

CONSTRAINED UNPREDICTABLE behavior is that PRSELR_EL2 register becomes UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the PRSELR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PRSELR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return PRSELR_EL2;
```

MSR PRSELR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRSELR_EL2 = X[t];
```

G1.3.28 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

There are no configuration notes.

Attributes

SCTLR_EL1 is a 64-bit register.

Field descriptions

The SCTLR_EL1 bit assignments are:
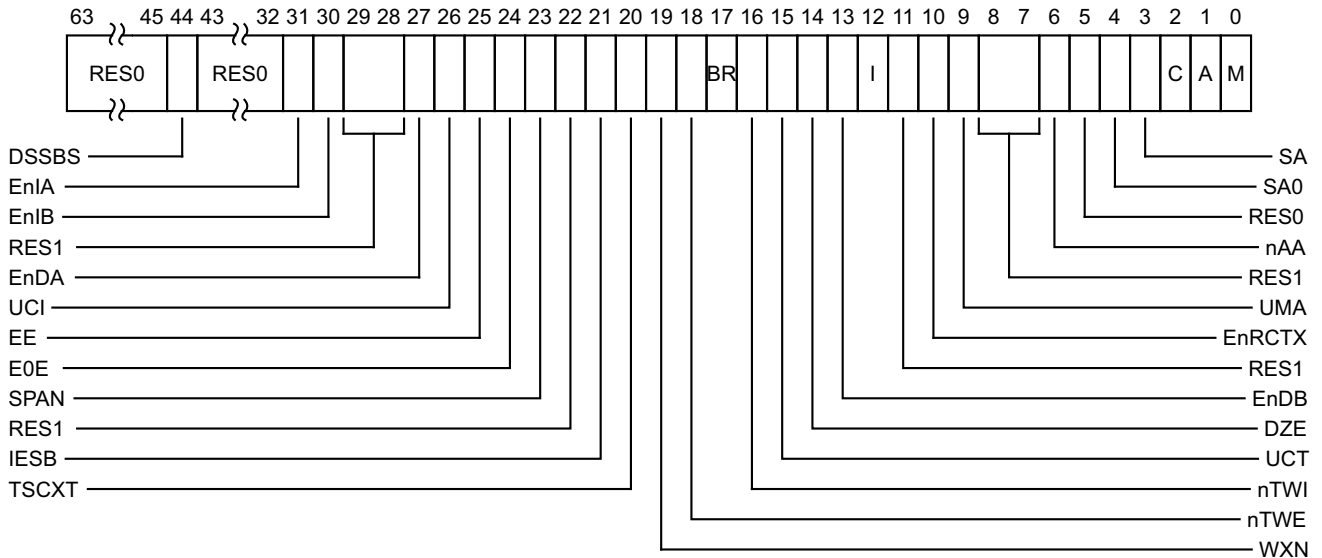
Bits [63:45]

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

0b0   PSTATE.SSBS is set to 0 on an exception to EL1.

0b1   PSTATE.SSBS is set to 1 on an exception to EL1.

In a system where the PE resets into EL1, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Bits [43:32]

Reserved, RES0.

EnIA, bit [31]

> ***When FEAT_PAuth is implemented:***
>
> > Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.
> >
> > For more information, see 'System register control of pointer authentication'.
> >
> > | | |
> > |---|---|
> > | 0b0 | Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled. |
> > | 0b1 | Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled. |
> >
> > —— **Note** ——
> >
> > This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.
> >
> > In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
> > Reserved, RES0.

EnIB, bit [30]

> ***When FEAT_PAuth is implemented:***
>
> > Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.
> >
> > For more information, see 'System register control of pointer authentication'.
> >
> > | | |
> > |---|---|
> > | 0b0 | Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled. |
> > | 0b1 | Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled. |
> >
> > —— **Note** ——
> >
> > This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.
> >
> > In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
> > Reserved, RES0.

Bits [29:28]

> > Reserved, RES1.

EnDA, bit [27]

> ***When FEAT_PAuth is implemented:***
>
> > Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.
> >
> > For more information, see 'System register control of pointer authentication'.
> >
> > | | |
> > |---|---|
> > | 0b0 | Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled. |
> > | 0b1 | Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled. |

——— **Note** ———

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

0b0   Execution of the specified instructions at EL0 using AArch64 is trapped.

0b1   This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

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:

0b0   Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

0b1   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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

The possible values of this bit are:

0b0   Explicit data accesses at EL0 are little-endian.

0b1   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 FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

SPAN, bit [23]

> ***When FEAT_PAN is implemented:***
>
> Set Privileged Access Never, on taking an exception to EL1.
>
> 0b0   PSTATE.PAN is set to 1 on taking an exception to EL1.
>
> 0b1   The value of PSTATE.PAN is left unchanged on taking an exception to EL1.
>
> When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.
>
> In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

> ***Otherwise:***
>
> Reserved, RES1.

Bit [22]

> Reserved, RES1.

IESB, bit [21]

> ***When FEAT_IESB is implemented:***
>
> Implicit Error Synchronization event enable. Possible values are:
>
> 0b0   Disabled.
>
> 0b1   An implicit error synchronization event is added:
>
> > • At each exception taken to EL1.
> >
> > • Before the operational pseudocode of each ERET instruction executed at EL1.
>
> When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL1 and before each DRPS instruction executed at EL1, in addition to the other cases where it is added.
>
> In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

> ***Otherwise:***
>
> Reserved, RES0.

TSCXT, bit [20]

> ***When FEAT_CSV2 is implemented:***
>
> Trap EL0 Access to the SCTXNUM_EL0 register, when EL0 is using AArch64. The defined values are:
>
> 0b0   EL0 access to SCTXNUM_EL0 is not disabled by this mechanism.
>
> 0b1   EL0 access to SCTXNUM_EL0 is disabled, causing an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

The value of SCTXNUM_EL0 is treated as 0.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

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:

0b0   This control has no effect on memory access permissions.

0b1   Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.

0b0   Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.

0b1   This control does not cause any instructions to be trapped.

——— **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 FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

BR, bit [17]

When VTCR_EL2.MSA == 0:

Background region enable for EL1 MPU memory regions.

0b0   Background region disabled for stage 1 EL1&0 translation regime.

0b1   Background region enabled for stage 1 EL1&0 translation regime.

If the value of HCR_EL2.{DC, TGE} is not {0, 0} then PE behaves as if the value of the SCTLR_EL1.BR field is 0 for all purposes other than returning the value of a direct read of the field.

If an EL1 MPU is enabled, then EL0 access that does not match the EL1 MPU region always results in a Translation fault.

In a system where the PE resets into EL1, this field resets to 0.

Otherwise:

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of `0x01`.

`0b0`   Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.

`0b1`   This control does not cause any instructions to be trapped.

———— **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 FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of `0x18`.

`0b0`   Accesses to the CTR_EL0 from EL0 using AArch64 are trapped.

`0b1`   This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of `0x18`.

`0b0`   Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped.

Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions this trap applies to are not supported.

`0b1`   This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see 'System register control of pointer authentication'.

`0b0`   Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.

`0b1`   Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

> ——— **Note** ———
>
> This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Stage 1 instruction access Cacheability control, for accesses at EL0 and EL1:

0b0   All instruction access to Stage 1 Normal memory from EL0 and EL1 are Stage 1 Non-cacheable.

If stage 1 EL1&0 translation is in VMSAv8-64 context and the value of SCTLR_EL1.M is 0, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

If stage 1 EL1&0 translation is in PMSAv8-64 context and the value of SCTLR_EL1.{BR, M} = {0, 0}, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

0b1   This control has no effect on the Stage 1 Cacheability of instruction access to Stage 1 Normal memory from EL0 and EL1.

If stage 1 EL1&0 translation is in VMSAv8-64 context and the value of SCTLR_EL1.M is 0, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

If stage 1 EL1&0 translation is in PMSAv8-64 context, and the value of SCTLR_EL1.{BR, M} = {0, 0}, then instruction accesses from stage 1 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 SCTLR_EL1.I bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to 0.

Bit [11]

Reserved, RES1.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Enable EL0 Access to the following instructions:

•   AArch64 CFP RCTX, DVP RCT and CPP RCTX instructions.

The defined values are:

0b0   EL0 access to these instructions is disabled, and these instructions are trapped to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

0b1   EL0 access to these instructions is enabled.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

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, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

0b0　　　　Any attempt at EL0 using AArch64 to execute an MRS, MSR(register), or MSR(immediate) instruction that accesses the DAIF is trapped.

0b1　　　　This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Bits [8:7]

Reserved, RES1.

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL1 and EL0 under certain conditions.

0b0　　　　LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for accesses.

0b1　　　　This control bit does not cause LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, or STLURH to generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Stage 1 Cacheability control, for data accesses.

0b0   All data access to Stage 1 Normal memory from EL0 and EL1, and all Normal memory accesses from unified cache to the EL1&0 Stage 1 translation tables, are treated as Stage 1 Non-cacheable.

0b1   This control has no effect on the Stage 1 Cacheability of:

• 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 SCTLR.C. This means that Non-secure EL0 and Non-secure EL1 data accesses to Normal memory are Cacheable.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0 .

0b0   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.

0b1   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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU or MPU enable for EL1&0 stage 1 address translation.

This is the enable bit for:

• MPU, if stage 1 EL1&0 translation is in PMSAv8-64 context.

• MMU, if stage 1 EL1&0 translation is in VMSAv8-64 context.

0b0   EL1 MPU (PMSAv8-64) or MMU(VMSAv8-64) disabled

See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.

0b1   EL1 MPU (PMSAv8-64) or MMU(VMSAv8-64) 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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

In a system where the PE resets into EL1, this field resets to 0.

Accessing the SCTLR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return SCTLR_EL1;
elsif PSTATE.EL == EL2 then
    return SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    SCTLR_EL1 = X[t];
```

G1.3.29 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.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, these controls apply also to execution at EL0.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions

The SCTLR_EL2 bit assignments are:

```
 63       45 44 43          32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10       7  6  5  4  3  2  1  0
┌─────────┬──┬──────────────┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──────────┬──┬──┬──┬──┬──┬──┬──┐
│  RES0   │  │    RES0      │  │  │  │  │  │  │  │  │  │  │  │BR│  │  │  │  │ I│  │  │   RES0   │  │  │  │  │ C│ A│ M│
└─────────┴──┴──────────────┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──────────┴──┴──┴──┴──┴──┴──┴──┘
```

DSSBS, EnIA, EnIB, RES1, EnDA, RES0, EE, RES0, RES1, IESB

SA, RES1, nAA, RES1, EnDB, RES0, RES1, RES1, WXN, RES0
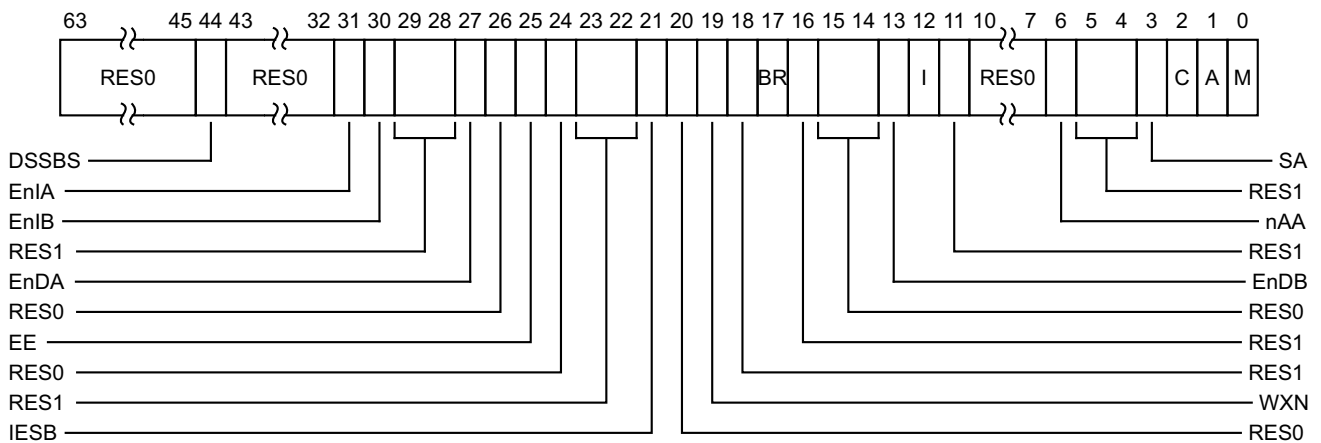
This format applies in all Armv8.0 implementations, and from Armv8.1 when the Effective value of HCR_EL2.{E2H, TGE} != {1, 1}.

Bits [63:45]

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

0b0 PSTATE.SSBS is set to 0 on an exception to EL2.

0b1 PSTATE.SSBS is set to 1 on an exception to EL2.

In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Bits [43:32]

Reserved, RES0.

EnIA, bit [31]

> ***When FEAT_PAuth is implemented:***
>
>> Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL2&0 translation regime.
>>
>> For more information, see 'System register control of pointer authentication'.
>>
>> | | |
>> |---|---|
>> | 0b0 | Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled. |
>> | 0b1 | Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled. |
>>
>> ──── **Note** ────
>>
>> This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.
>>
>> ────────────────
>>
>> In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

EnIB, bit [30]

> ***When FEAT_PAuth is implemented:***
>
>> Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL2&0 translation regime.
>>
>> For more information, see 'System register control of pointer authentication'.
>>
>> | | |
>> |---|---|
>> | 0b0 | Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled. |
>> | 0b1 | Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled. |
>>
>> ──── **Note** ────
>>
>> This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.
>>
>> ────────────────
>>
>> In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

Bits [29:28]

> Reserved, RES1.

EnDA, bit [27]

> ***When FEAT_PAuth is implemented:***
>
>> Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL2&0 translation regime.
>>
>> For more information, see 'System register control of pointer authentication'.
>>
>> | | |
>> |---|---|
>> | 0b0 | Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled. |
>> | 0b1 | Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled. |

———— **Note** ————

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

————————————

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

EE, bit [25]

Endianness of data accesses at EL2.

0b0   Explicit data accesses at EL2 are little-endian.

0b1   Explicit data accesses at EL2 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 a system where the PE resets into EL2, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable.

0b0   Disabled.

0b1   An implicit error synchronization event is added:

•   At each exception taken to EL2.

•   Before the operational pseudocode of each ERET instruction executed at EL2.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL2 and before each DRPS instruction executed at EL2, in addition to the other cases where it is added.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN:

0b0   This control has no effect on memory access permissions.

0b1   Any region that is writable in the EL2 or EL2&0 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

BR, bit [17]

Background region enable for EL2 MPU memory regions.

0b0   Background region disabled for stage 1 EL2 translation regime and stage 2 EL1&0 translation regime.

0b1   Background region enabled for stage 1 EL2 translation regime and stage 2 EL1&0 translation regime.

If EL2 MPU is enabled, then EL0 and EL1 access that does not match an EL2 MPU region always results in a Translation fault.

In a system where the PE resets into EL2, this field resets to 0.

Bit [16]

Reserved, RES1.

Bits [15:14]

Reserved, RES0.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL2&0 translation regime.

For more information, see 'System register control of pointer authentication'.

0b0   Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.

0b1   Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

——— **Note** ———

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2:

0b0   All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL2.{BR, M} = {0, 0}, then instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

0b1　　　　This control has no effect on the Cacheability of instruction access to Normal memory from EL2.

If the value of SCTLR_EL2.{BR, M} = {0, 0}, then 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.

In a system where the PE resets into EL2, this field resets to 0.

Bit [11]

Reserved, RES1.

Bits [10:7]

Reserved, RES0.

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL2 under certain conditions.

0b0　　　　LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for accesses.

0b1　　　　This control bit does not cause LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH, STLLR, STLLRH, STLR, STLRH, STLUR, or STLURH to generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

0b0　　　　All data accesses to Normal memory from EL2 are Non-cacheable for all levels of data and unified cache.

0b1　　　　This control has no effect on the Cacheability of:

　　　　•　　Data access to Normal memory from EL2.

This bit has no effect on the EL1&0 or EL3 translation regimes.

In a system where the PE resets into EL2, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

0b0   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.

0b1   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.

In a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MPU enable for EL2 stage 1 and EL1&0 stage 2 address translation.

0b0   MPU disabled for EL2 and EL1&0 stage 2 address translation.

See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.

0b1   MPU enabled for EL2 and EL1&0 stage 2 address translation.

In a system where the PE resets into EL2, this field resets to 0.

Accessing the SCTLR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, SCTLR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SCTLR_EL2;
```

MSR SCTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    SCTLR_EL2 = X[t];
```

MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return SCTLR_EL1;
elsif PSTATE.EL == EL2 then
    return SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    SCTLR_EL1 = X[t];
```

G1.3.30   TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

> The control register for stage 1 of the EL1&0 translation regime.

Configurations

> In a PMSAv8-64 only implementation, this register is UNDEFINED.

Attributes

> TCR_EL1 is a 64-bit register.

Field descriptions

The TCR_EL1 bit assignments are:
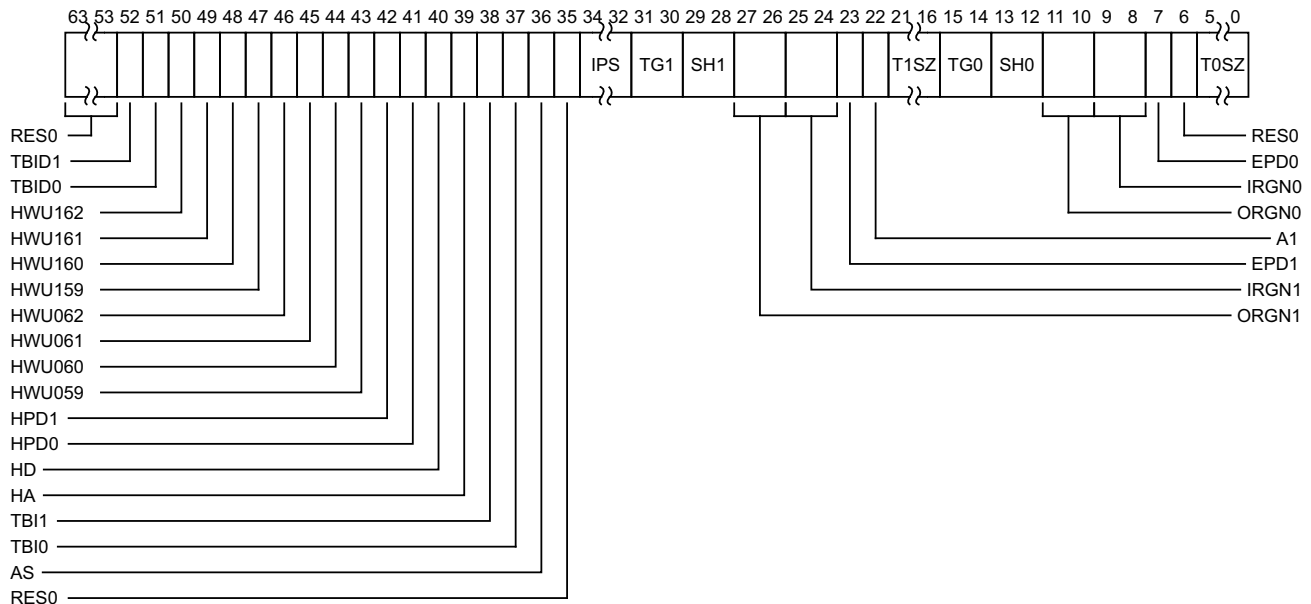
Any of the bits in TCR_EL1, other than the A1 bit and the EPDx bits when they have the value 1, are permitted to be cached in a TLB.

Bits [63:53]

> Reserved, RES0.

TBID1, bit [52]

> *When VTCR_EL2.MSA == 1 and FEAT_PAuth is implemented:*
>
> Controls the use of the top byte of instruction addresses for address matching.
>
> For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.
>
> For more information, see 'Address tagging in AArch64 state'.
>
> 0b0   TCR_EL1.TBI1 applies to Instruction and Data accesses.
>
> 0b1   TCR_EL1.TBI1 applies to Data accesses only.
>
> This affects addresses where the address would be translated by tables pointed to by TTBR1_EL1.
>
> This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0 and FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

Reads as 0b0.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When VTCR_EL2.MSA == 1 and FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging in AArch64 state'.

0b0   TCR_EL1.TBI0 applies to Instruction and Data accesses.

0b1   TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL1.

This field resets to an architecturally UNKNOWN value.

When FEAT_PAuth is implemented and VTCR_EL2.MSA == 0:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging in AArch64 state'.

0b0   TCR_EL1.TBI0 applies to Instruction and Data accesses.

0b1   TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by EL1 MPU.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

0b0   For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1   For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

0b0　　For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1　　For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

0b0　　For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1　　For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

0b0　　For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1　　For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

> *When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:*
>
>> Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.
>>
>> 0b0   For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
>>
>> 0b1   For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.
>>
>> The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> *When VTCR_EL2.MSA == 0:*
>
>> Reserved, RES0.
>
> *Otherwise:*
>
>> Reserved, RES0.

HWU061, bit [45]

> *When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:*
>
>> Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.
>>
>> 0b0   For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
>>
>> 0b1   For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.
>>
>> The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> *When VTCR_EL2.MSA == 0:*
>
>> Reserved, RES0.
>
> *Otherwise:*
>
>> Reserved, RES0.

HWU060, bit [44]

> *When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:*
>
>> Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.
>>
>> 0b0   For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
>>
>> 0b1   For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.
>>
>> The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> *When VTCR_EL2.MSA == 0:*
>
>> Reserved, RES0.
>
> *Otherwise:*
>
>> Reserved, RES0.

HWU059, bit [43]

When VTCR_EL2.MSA == 1 and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

0b0   For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1   For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When VTCR_EL2.MSA == 1 and FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL1.

0b0   Hierarchical permissions are enabled.

0b1   Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When VTCR_EL2.MSA == 1 and FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL1.

0b0   Hierarchical permissions are enabled.

0b1   Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HD, bit [40]

When VTCR_EL2.MSA == 1 and FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL0 and EL1.

0b0   Stage 1 hardware management of dirty state disabled.

0b1   Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

HA, bit [39]

When VTCR_EL2.MSA == 1 and FEAT_HAFDBS is implemented:

Hardware Access flag update in stage 1 translations from EL0 and EL1.

0b0   Stage 1 Access flag update disabled.

0b1   Stage 1 Access flag update enabled.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

When VTCR_EL2.MSA == 1:

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.

0b0   Top Byte used in the address calculation.

0b1   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.

If FEAT_PAuth is implemented and TCR_EL1.TBID1 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

•   A branch or procedure return within EL0 or EL1.

•   An exception taken to EL1.

•   An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Top Byte ignored.

Reads as 0b0.

Otherwise:

Reserved, RES0.

TBI0, bit [37]

When VTCR_EL2.MSA == 1:

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.

0b0   Top Byte used in the address calculation.

0b1   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.

If FEAT_PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.

- An exception taken to EL1.

- An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Top Byte ignored. Indicates whether the top byte of an address is used for an address match for the EL1 MPU regions, or ignored and used for tagged addresses.

0b0   Top Byte used in the address calculation.

0b1   Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1, where the address would be translated by the EL1 MPU.

If FEAT_PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.

- An exception taken to EL1.

- An exception return to EL0 or EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AS, bit [36]

When VTCR_EL2.MSA == 1:

ASID Size.

0b0   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.

0b1   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.

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

ASID Size.

0b0   8 bit - the upper 8 bits of TTBR0_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros when used for address matching.

0b1   16 bit - the upper 16 bits of TTBR0_EL1 are used for address matching.

If the implementation has only 8 bits of ASID, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

When VTCR_EL2.MSA == 1:

Intermediate Physical Address Size.

0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.

All other values are reserved.

The reserved values behave in the same way as the 0b101 or 0b110 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.

If the translation granule is not 64KB, the value 0b110 is treated as reserved.

It is IMPLEMENTATION DEFINED whether an implementation that does not implement FEAT_LPA supports setting the value of 0b110 for the 64KB translation granule size or whether setting this value behaves as the 0b101 encoding.

In an implementation that supports 52-bit PAs, if the value of this field is not 0b110 or a value treated as 0b110, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL1 are 0b0000.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TG1, bits [31:30]

When VTCR_EL2.MSA == 1:

Granule size for the TTBR1_EL1.

0b01	16KB.
0b10	4KB.
0b11	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.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH1, bits [29:28]

> ***When VTCR_EL2.MSA == 1:***
>
>> Shareability attribute for memory associated with translation table walks using TTBR1_EL1.
>>
>> 0b00   Non-shareable.
>>
>> 0b10   Outer Shareable.
>>
>> 0b11   Inner Shareable.
>>
>> Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

ORGN1, bits [27:26]

> ***When VTCR_EL2.MSA == 1:***
>
>> Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL1.
>>
>> 0b00   Normal memory, Outer Non-cacheable.
>>
>> 0b01   Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
>>
>> 0b10   Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
>>
>> 0b11   Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

IRGN1, bits [25:24]

> ***When VTCR_EL2.MSA == 1:***
>
>> Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL1.
>>
>> 0b00   Normal memory, Inner Non-cacheable.
>>
>> 0b01   Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
>>
>> 0b10   Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
>>
>> 0b11   Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

EPD1, bit [23]

> ***When VTCR_EL2.MSA == 1:***
>
>> 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:
>>
>> 0b0   Perform translation table walks using TTBR1_EL1.
>>
>> 0b1   A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

A1, bit [22]

> ***When VTCR_EL2.MSA == 1:***
>
>> Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:
>>
>> 0b0   TTBR0_EL1.ASID defines the ASID.
>>
>> 0b1   TTBR1_EL1.ASID defines the ASID.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

T1SZ, bits [21:16]

> ***When VTCR_EL2.MSA == 1:***
>
>> 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.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

TG0, bits [15:14]

> ***When VTCR_EL2.MSA == 1:***
>
>> Granule size for the TTBR0_EL1.
>>
>> 0b00   4KB
>>
>> 0b01   64KB
>>
>> 0b10   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.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

SH0, bits [13:12]

> ***When VTCR_EL2.MSA == 1:***
>
>> Shareability attribute for memory associated with translation table walks using TTBR0_EL1.
>>
>> 0b00   Non-shareable
>>
>> 0b10   Outer Shareable
>>
>> 0b11   Inner Shareable
>>
>> Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.
>>
>> This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
>> Reserved, RES0.

ORGN0, bits [11:10]

> ***When VTCR_EL2.MSA == 1:***

>> Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

>> | | |
>> |---|---|
>> | 0b00 | Normal memory, Outer Non-cacheable. |
>> | 0b01 | Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable. |
>> | 0b10 | Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable. |
>> | 0b11 | Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable. |

>> This field resets to an architecturally UNKNOWN value.

> ***Otherwise:***

>> Reserved, RES0.

IRGN0, bits [9:8]

> ***When VTCR_EL2.MSA == 1:***

>> Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

>> | | |
>> |---|---|
>> | 0b00 | Normal memory, Inner Non-cacheable. |
>> | 0b01 | Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable. |
>> | 0b10 | Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable. |
>> | 0b11 | Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable. |

>> This field resets to an architecturally UNKNOWN value.

> ***Otherwise:***

>> Reserved, RES0.

EPD0, bit [7]

> ***When VTCR_EL2.MSA == 1:***

>> 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:

>> | | |
>> |---|---|
>> | 0b0 | Perform translation table walks using TTBR0_EL1. |
>> | 0b1 | A TLB miss on an address that is translated using TTBR0_EL1 generates a Translation fault. No translation table walk is performed. |

>> This field resets to an architecturally UNKNOWN value.

> ***Otherwise:***

>> Reserved, RES0.

Bit [6]

>> Reserved, RES0.

T0SZ, bits [5:0]

> ***When VTCR_EL2.MSA == 1:***

>> The size offset of the memory region addressed by TTBR0_EL1. The region size is $2^{(64-\text{T0SZ})}$ 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.

>> This field resets to an architecturally UNKNOWN value.

> ***Otherwise:***

>> Reserved, RES0.

Accessing the TCR_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TCR_EL1;
elsif PSTATE.EL == EL2 then
    return TCR_EL1;
```

MSR TCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    TCR_EL1 = X[t];
```

G1.3.31 TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

Purpose

The control register for stage 1 of the EL2, or EL2&0, translation regime:

- When the Effective value of HCR_EL2.E2H is 0, this register controls stage 1 of the EL2 translation regime, that supports a single VA range, translated using TTBR0_EL2.

- When the value of HCR_EL2.E2H is 1, this register controls stage 1 of the EL2&0 translation regime, that supports both:

  — A lower VA range, translated using TTBR0_EL2.

  — A higher VA range, translated using TTBR1_EL2.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

TCR_EL2 is a 64-bit register.

Field descriptions

The TCR_EL2 bit assignments are:
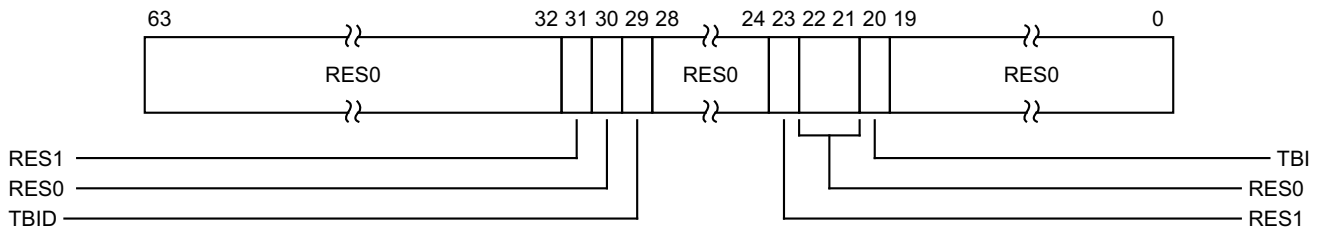
Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

TBID, bit [29]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging in AArch64 state'.

0b0 TCR_EL2.TBI applies to Instruction and Data accesses.

0b1 TCR_EL2.TBI applies to Data accesses only.

This affects addresses where the address would be translated by EL2 MPU.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [28:24]

Reserved, RES0.

Bit [23]

Reserved, RES1.

Bits [22:21]

Reserved, RES0.

TBI, bit [20]

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the EL2 MPU regions, or ignored and used for tagged addresses.

For more information, see 'Address tagging in AArch64 state'.

0b0   Top Byte used in the address calculation.

0b1   Top Byte ignored in the address calculation.

This affects addresses generated in EL2 using AArch64 where the address would be translated by the EL2 MPU. It has an effect whether the EL2 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID is 1, then this field only applies to Data accesses.

If the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

•   A branch or procedure return within EL2.

•   An exception taken to EL2.

•   An exception return to EL2.

This field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Accessing the TCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return TCR_EL2;
```

MSR TCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    TCR_EL2 = X[t];
```

MRS <Xt>, TCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return TCR_EL1;
elsif PSTATE.EL == EL2 then
    return TCR_EL1;
```

MSR TCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    TCR_EL1 = X[t];
```

G1.3.32 TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

Purpose

> Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL1&0 translation regime, and other information for this translation regime.

Configurations

> There are no configuration notes.

Attributes

> TTBR0_EL1 is a 64-bit register.

Field descriptions

The TTBR0_EL1 bit assignments are:
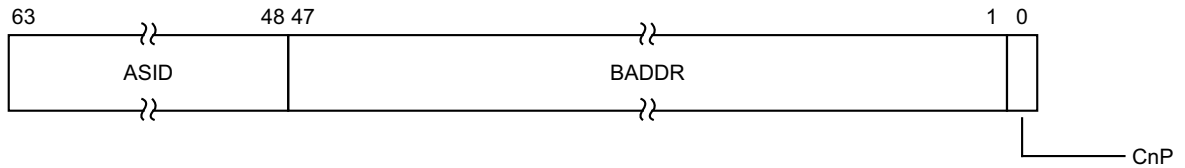
ASID, bits [63:48]

> ***When VTCR_EL2.MSA == 1:***
>
> > 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.
> >
> > This field resets to an architecturally UNKNOWN value.
>
> ***When VTCR_EL2.MSA == 0:***
>
> > An ASID for addresses defined by the current EL1 MPU configuration.
> >
> > If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.
> >
> > This field resets to an architecturally UNKNOWN value.
>
> ***Otherwise:***
>
> > Reserved, RES0.

BADDR, bits [47:1]

> ***When VTCR_EL2.MSA == 1:***
>
> > Translation table base address, A[47:x] or A[51:x], bits[47:1].
>
> > ──── **Note** ────
> >
> > • Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes FEAT_LPA and is using the 64KB translation granule.
> >
> > • A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes FEAT_LPA, if the value of TCR_EL1.IPS is 0b110, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  — If x >= 6 then z=x.
  — Otherwise, z=6.
- Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes FEAT_TTCNP bit[0] of the stage 1 translation table base address is zero.

—— **Note** ——

- In an implementation that includes FEAT_LPA a TCR_EL1.IPS value of 0b110, that selects an IPA size of 52 bits, is permitted only when using the 64KB translation granule.
- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the Effective value of TCR_EL1.IPS is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of TCR_EL1.IPS is not 0b110 then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are 0b0000.

—— **Note** ——

This definition applies:

- To an implementation that includes FEAT_LPA and is using a translation granule smaller than 64KB.
- To any implementation that does not include FEAT_LPA.

If any TTBR0_EL1[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR0_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register 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.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CnP, bit [0]

When VTCR_EL2.MSA == 1 and FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1.

0b0   The translation table entries pointed to by TTBR0_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR0_EL1 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR0_EL1.CnP on those other PEs.

- The value of the current ASID.

- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

0b1   The translation table entries pointed to by TTBR0_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR0_EL1.

- The translation tables relate to the same translation regime.

- The ASID is the same as the current ASID.

- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This field is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

——— **Note** ———

If the value of the TTBR0_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

———————————

This field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == 0:

Reserved, RES0.

Otherwise:

Reserved, RES0.

Accessing the TTBR0_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
```

```
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                return TTBR0_EL1;
    elsif PSTATE.EL == EL2 then
        return TTBR0_EL1;
```

MSR TTBR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TTBR0_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    TTBR0_EL1 = X[t];
```

G1.3.33 VSCTLR_EL2, Virtualization System Control Register (EL2)

The VSCTLR_EL2 characteristics are:

Purpose

Provides configuration information for VMSAv8-64 and PMSAv8-64 virtualization using stage 2 of EL1&0 translation regime.

Configurations

There are no configuration notes.

Attributes

VSCTLR_EL2 is a 64-bit register.

Field descriptions

The VSCTLR_EL2 bit assignments are:
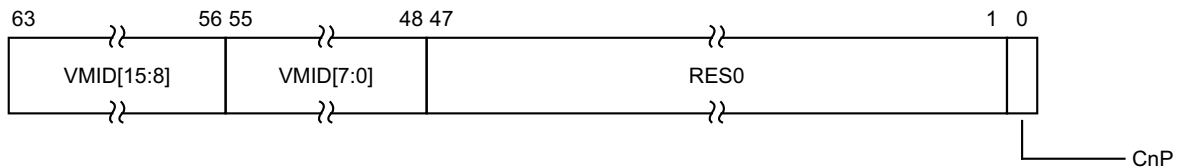
VMID[15:8], bits [63:56]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. See VSCTLR_EL2.VMID[7:0] for more details.

If the implementation has an 8-bit VMID, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [55:48]

The VMID for the EL1-Guest-OS.

The VMID is 8 bits when any of the following are true:

- The VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

This field resets to an architecturally UNKNOWN value.

Bits [47:1]

Reserved, RES0.

CnP, bit [0]

When FEAT_TTCNP is implemented and VTCR_EL2.MSA == 1:

Common not Private. This bit indicates whether stage 2 of EL1&0 translations are a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VSCTLR_EL2.CnP is 1.

0b0   The stage 2 translations of the EL1&0 translation regime are permitted to differ in other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.

0b1   The stage 2 translations of the EL1&0 translation regime are the same for every other PE in the Inner Shareable domain for which the value of VSCTLR_EL2.CnP is 1 and the VMID is the same as the current VMID.

——— **Note** ———

If the value of VSCTLR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and the stage 2 EL1&0 translation does not point to the same configurations when using the current VMID, then the results of the translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

In an implementation that does not support VMSAv8-64 at stage 1 EL1&0 translation regime this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the VSCTLR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VSCTLR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VSCTLR_EL2;
```

MSR VSCTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VSCTLR_EL2 = X[t];
```

G1.3.34   VSTCR_EL2, Virtualization Secure Translation Control Register

The VSTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the Secure EL1&0 translation regime.

Configurations

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to VSTCR_EL2 are UNDEFINED.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VSTCR_EL2 is a 64-bit register.

Field descriptions

The VSTCR_EL2 bit assignments are:
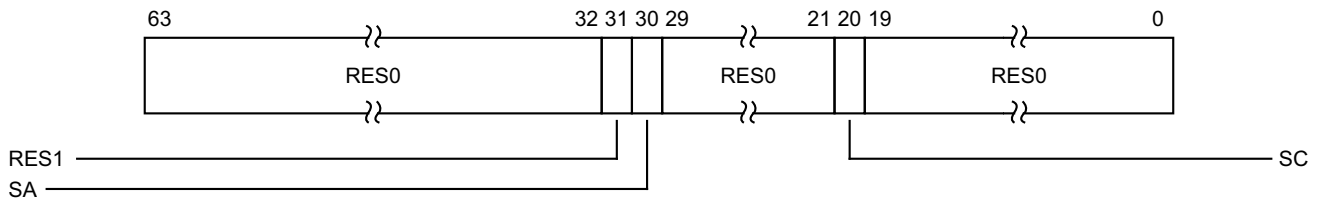
Any of the bits in VSTCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

SA, bit [30]

Secure stage 2 translation output address space.

0b0   All stage 2 translations for the Secure PA space access the Secure PA space.

0b1   All stage 2 translations for the Secure PA space access the Non-secure PA space.

This field resets to an architecturally UNKNOWN value.

Bits [29:21]

Reserved, RES0.

SC, bit [20]

NS check enable bit.

0b0   Least secure NS configuration is selected from the stage 1 and stage 2 EL1&0 translation regime for the given address.

0b1   Stage 2 NS configuration is checked against stage 1 NS configuration in EL1&0 translation regime for the given address, and generate a fault if they are different.

This field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Accessing the VSTCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VSTCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VSTCR_EL2;
```

MSR VSTCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VSTCR_EL2 = X[t];
```

G1.3.35 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the EL1&0 translation regime.

Configurations

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VTCR_EL2 is a 64-bit register.

Field descriptions

The VTCR_EL2 bit assignments are:
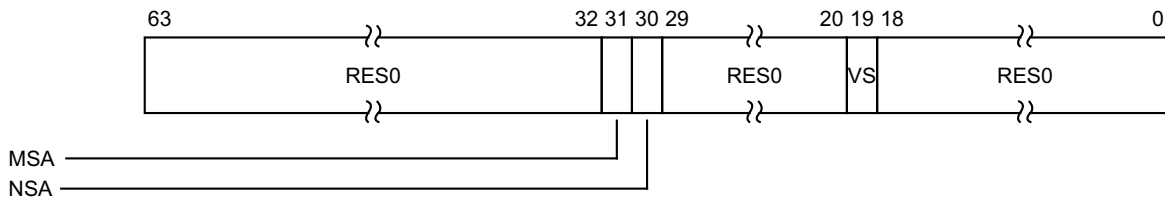
Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

MSA, bit [31]

When ID_AA64MMFR0_EL1.MSA_frac == 0b0010:

Stage 1 EL1&0 translation regime memory system architecture.

0b0   Stage 1 EL1&0 translation regime uses PMSAv8-64 memory architecture.

0b1   Stage 1 EL1&0 translation regime uses VMSAv8-64 memory architecture.

This field resets to an architecturally UNKNOWN value.

When ID_AA64MMFR0_EL1.MSA_frac == 0b0001:

Reserved, RES0.

Otherwise:

Reserved, RES1.

NSA, bit [30]

When FEAT_SEL2 is implemented:

Non-secure stage 2 translation output address space.

0b0   All stage 2 translations for the Non-secure PA space of the Secure EL1&0 translation regime access the Secure PA space.

0b1   All stage 2 translations for the Non-secure PA space of the Secure EL1&0 translation regime access the Non-secure PA space.

This bit behaves as 1 for all purposes other than reading back the value of the bit when the value of VSTCR_EL2.SA is 1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:20]

Reserved, RES0.

VS, bit [19]

When FEAT_VMID16 is implemented:

VMID Size.

0b0   8 bit - the upper 8 bits of VSCTLR_EL2 is ignored by the hardware, and treated as if they are all zeros, for every purpose except when reading back the register.

0b1   16 bit - the upper 8 bits of VSCTLR_EL2 is used for allocation and matching in the TLB.

If the implementation only supports an 8-bit VMID, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [18:0]

Reserved, RES0.

Accessing the VTCR_EL2

Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, VTCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return VTCR_EL2;
```

MSR VTCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTCR_EL2 = X[t];
```

G1.4   Debug registers

This section lists the Debug System registers in Armv8-R AArch64, in alphabetical order.

G1.4.1   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[31:0] is architecturally mapped to External register
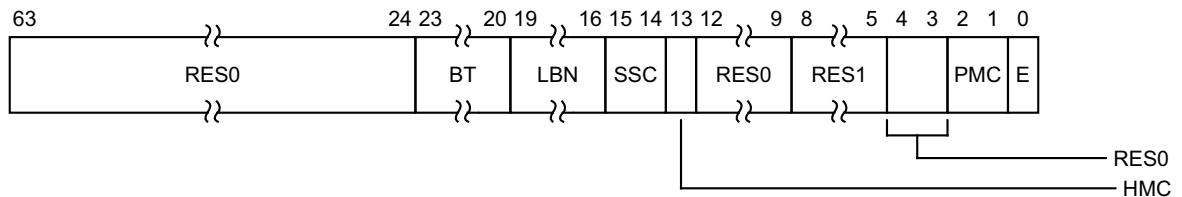>DBGBCR<n>_EL1[31:0].

>If breakpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

>DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions

The DBGBCR<n>_EL1 bit assignments are:

Bits [63:24]

>Reserved, RES0.

BT, bits [23:20]

>Breakpoint Type. Possible values are:

>0b0000   Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an
>instruction.

>0b0001   As 0b0000, but linked to a Context matching breakpoint.

>0b0010   Unlinked Context ID match. When FEAT_VHE is implemented, EL2 is using
>AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing
>at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then
>DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL2 value. Otherwise,
>DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL1 value

>0b0011   As 0b0010, with linking enabled.

>0b0110   Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID
>compared against CONTEXTIDR_EL1.

>0b0111   As 0b0110, with linking enabled.

>0b1000   Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against
>VSCTLR_EL2.VMID.

>0b1001   As 0b1000, with linking enabled.

>0b1010   Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID
>compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID
>compared against VSCTLR_EL2.VMID.

>0b1011   As 0b1010, with linking enabled.

0b1100	Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.
0b1101	As 0b1100, with linking enabled.
0b1110	Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.
0b1111	As 0b1110, with linking enabled.

All other values are reserved. Constraints on breakpoint programming mean other values are reserved under some conditions.

For more information on the operation of the SSC, HMC, and PMC fields, and on the effect of programming this field to a reserved value, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions' and 'Reserved DBGBCR<n>_EL1.BT values'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On a Warm reset, the value of this field is unchanged.

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.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On a Warm reset, the value of this field is unchanged.

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 on the operation of the SSC, HMC, and PMC fields, and the effect of programming the fields to a reserved set of values, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions' and 'Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On a Warm reset, the value of this field is unchanged.

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.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On a Warm reset, the value of this field is unchanged.

Bits [12:9]

Reserved, RES0.

Bits [8:5]

Reserved, RES1.

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.

> For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

> On a Cold reset, this field resets to an architecturally UNKNOWN value.

> On a Warm reset, the value of this field is unchanged.

E, bit [0]

> Enable breakpoint DBGBVR<n>_EL1.

> 0b0   Breakpoint disabled.

> 0b1   Breakpoint enabled.

> On a Cold reset, this field resets to an architecturally UNKNOWN value.

> On a Warm reset, the value of this field is unchanged.

Accessing the DBGBCR<n>_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, DBGBCR<n>_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR_EL1[UInt(CRm<3:0>)];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        return DBGBCR_EL1[UInt(CRm<3:0>)];
```

MSR DBGBCR<n>_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	n[3:0]	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
```

```
                    DBGBCR_EL1[UInt(CRm<3:0>)] = X[t];
            elsif PSTATE.EL == EL2 then
                if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
                    Halt(DebugHalt_SoftwareAccess);
                else
                    DBGBCR_EL1[UInt(CRm<3:0>)] = X[t];
```

G1.4.2 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

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

MDCR_EL2 is a 64-bit register.

Field descriptions

The MDCR_EL2 bit assignments are:
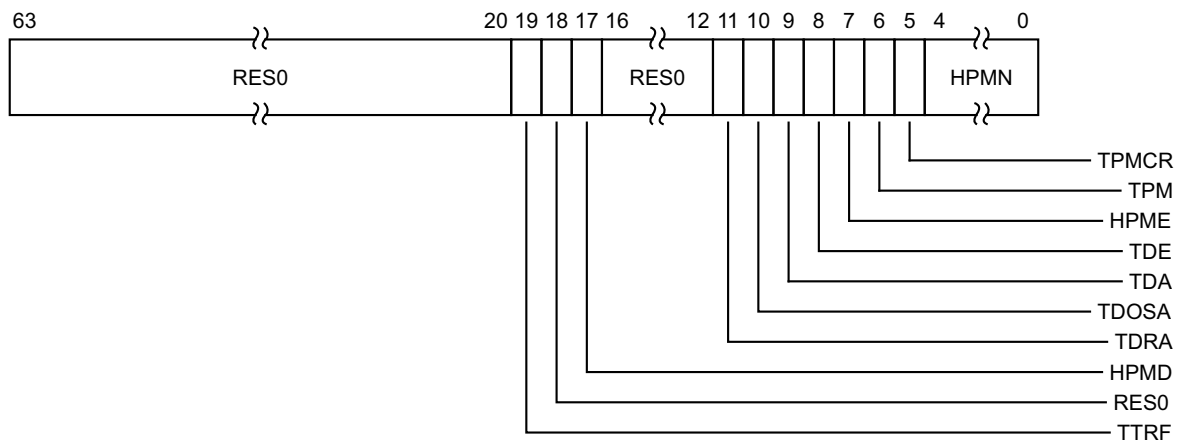
Bits [63:20]

Reserved, RES0.

TTRF, bit [19]

When FEAT_TRF is implemented:

Traps use of the Trace Filter Control registers at EL1 to EL2, as follows:

- Access to TRFCR_EL1 is trapped to EL2, reported using EC syndrome value 0x18.

0b0   Accesses to TRFCR_EL1 and TRFCR at EL1 are not affected by this control.

0b1   Accesses to TRFCR_EL1 and TRFCR at EL1 generate a trap exception to EL2 when EL2 is enabled in the current Security state.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

When FEAT_PMUv3p1 is implemented:

Guest Performance Monitors Disable. This control prohibits event counting at EL2.

0b0   Event counting allowed at EL2.

0b1   Event counting prohibited at EL2.

        If FEAT_Debugv8p2 is not implemented, event counting is prohibited unless enabled by the IMPLEMENTATION DEFINED authentication interface ExternalSecureNoninvasiveDebugEnabled().

This control applies only to:

- The event counters in the range [0..(MDCR_EL2.HPMN-1)].

- 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.

On a Warm reset, this field resets to 0.

Otherwise:

Reserved, RES0.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps System register accesses to the Debug ROM registers to EL2 when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64 state, accesses to MDRAR_EL1 are trapped to EL2, reported using EC syndrome value 0x18.

0b0   This control does not cause any instructions to be trapped.

0b1   EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by DBGDSCRext.UDCCdis or MDSCR_EL1.TDCC.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.

- HCR_EL2.TGE == 1.

———— **Note** ————

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDOSA, bit [10]

When FEAT_DoubleLock is implemented:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:

    — OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, and DBGPRCR_EL1.

    — Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

0b0   This control does not cause any instructions to be trapped.

0b1   EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

——— **Note** ———

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:

    — OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.

    — Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 are trapped.

0b0   This control does not cause any instructions to be trapped.

0b1   EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

——— **Note** ———

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

——— **Note** ———

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap Debug Access. Traps EL0 and EL1 System register accesses to debug System registers that are not trapped by MDCR_EL2.TDRA or MDCR_EL2.TDOSA, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 reported using EC syndrome value 0x18:

    — MDCCSR_EL0, MDCCINT_EL1, OSDTRRX_EL1, MDSCR_EL1, OSDTRTX_EL1, OSECCR_EL1, DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_El1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, DBGAUTHSTATUS_EL1.

    — When not in Debug state, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0.

0b0   This control does not cause any instructions to be trapped.

0b1   EL0 or EL1 System register accesses to the debug registers are trapped from both Execution states to EL2 when EL2 is enabled in the current Security state, unless the access generates a higher priority exception.

Traps of AArch64 accesses to DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0 are ignored in Debug state.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1
- HCR_EL2.TGE == 1

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDE, bit [8]

Trap Debug Exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL_D.

0b0   The debug target Exception level is EL1.

0b1   If EL2 is enabled for the current Effective value of SCR_EL3.NS, the debug target Exception level is EL2, otherwise the debug target Exception level is EL1.

              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 the register.

For more information, see 'Routing debug exceptions'.

This field is treated as being 1 for all purposes other than a direct read when HCR_EL2.TGE == 1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

HPME, bit [7]

When FEAT_PMUv3 is implemented:

[MDCR_EL2.HPMN..(N-1)] event counters enable.

0b0   Event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] are disabled.

0b1   Event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] are enabled by PMCNTENSET_EL0.

If MDCR_EL2.HPMN is less than PMCR_EL0.N or PMCR.N, the event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] or [HDCR.HPMN..(PMCR.N-1)], are enabled and disabled by this bit. Otherwise this bit has no effect on the operation of the event counters.

———— **Note** ————

The effect of MDCR_EL2.HPMN on the operation of this bit applies regardless of whether EL2 is enabled in the current Security state.

————————————————

For more information see the description of the HPMN field.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When FEAT_PMUv3 is implemented:

Trap Performance Monitors accesses. Traps EL0 and EL1 accesses to all Performance Monitor registers to EL2 when EL2 is enabled in the current Security state, from both Execution states, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:

  — PMCR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVSCLR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0.

  — If FEAT_PMUv3p4 is implemented, PMMIR_EL1

0b0   This control does not cause any instructions to be trapped.

0b1   EL0 and EL1 accesses to all Performance Monitor registers are trapped to EL2 when EL2 is enabled in the current Security state.

——— **Note** ———

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPMCR, bit [5]

When FEAT_PMUv3 is implemented:

Trap PMCR_EL0 or PMCR accesses. Traps EL0 and EL1 accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to PMCR_EL0 are trapped to EL2, reported using EC syndrome value 0x18.

0b0   This control does not cause any instructions to be trapped.

0b1   EL0 and EL1 accesses to the PMCR_EL0 or PMCR are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by PMUSERNR.EN or PMUSERNR_EL0.EN.

——— **Note** ———

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPMN, bits [4:0]

When FEAT_PMUv3 is implemented:

Defines the number of event counters that are accessible from EL3, EL2, EL1, and from EL0 if permitted.

If HPMN is less than PMCR_EL0.N, HPMN divides the Performance Monitors into two ranges: [0..(HPMN-1)] and [HPMN..(PMCR_EL0.N-1)].

For an event counter in the range [0..(HPMN-1)]:

- The counter is accessible from EL3, EL2, and EL1, and from EL0 if permitted by PMUSERNR_EL0 or PMUSERNR.

- The counter is enabled by PMCR_EL0.E or PMCR.E and bit <n> of PMCNTENSET_EL0.

—— **Note** ——

If HPMN is equal to PMCR_EL0.N, this applies to all event counters.

If HPMN is less than PMCR_EL0.N, for an event counter in the range [HPMN..(PMCR_EL0.N-1)]:

- The counter is accessible from EL2 and EL3.

- If FEAT_SEL2 is disabled or is not implemented, the counter is also accessible from Secure EL1, and from Secure EL0 if permitted by PMUSERNR_EL0.

- The counter is enabled by MDCR_EL2.HPME or HDCR.HPME and bit <n> of PMCNTENSET_EL0.

If this field is set to 0, or to a value larger than PMCR_EL0.N, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.

- Either:

  — An UNKNOWN number of counters are reserved for EL2 and EL3 use. That is, the PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN non-zero value less than or equal to PMCR_EL0.N.

  — All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

On a Warm reset, this field resets to the value in PMCR_EL0.N.

Otherwise:

Reserved, RES0.

Accessing the MDCR_EL2

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return MDCR_EL2;
```

MSR MDCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    MDCR_EL2 = X[t];
```

G1.4.3 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

Purpose

Main control register for the debug implementation.

Configurations

There are no configuration notes.

Attributes

MDSCR_EL1 is a 64-bit register.

Field descriptions

The MDSCR_EL1 bit assignments are:
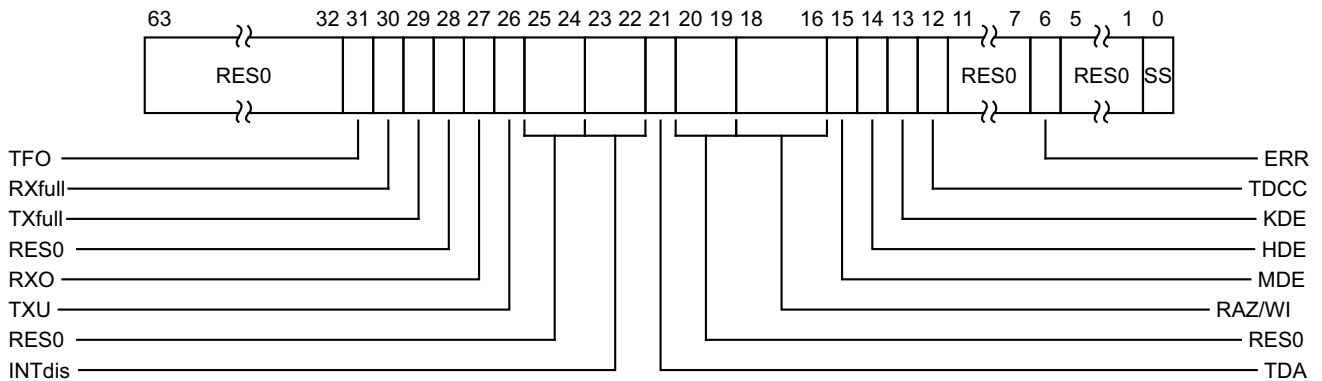
Bits [63:32]

Reserved, RES0.

TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter override. Used for save/restore of EDSCR.TFO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

Accessing this field has the following behavior:
- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

Used for save/restore of EDSCR.RXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

TXfull, bit [29]

Used for save/restore of EDSCR.TXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of EDSCR.INTdis.

When OSLSR_EL1.OSLK == 0, and software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

TDA, bit [21]

Used for save/restore of EDSCR.TDA.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bits [20:19]

Reserved, RES0.

Bits [18:16]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

MDE, bit [15]

Monitor debug events. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

0b0   Breakpoint, Watchpoint, and Vector Catch exceptions disabled.

0b1   Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

KDE, bit [13]

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D. Permitted values are:

0b0   Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.

0b1   All debug exceptions enabled within EL_D.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDCC, bit [12]

Traps EL0 accesses to the Debug Communication Channel (DCC) registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, as follows:

- In AArch64 state, MRS or MSR accesses to the following DCC registers are trapped, reported using EC syndrome value 0x18:
  - MDCCSR_EL0.
  - If not in Debug state, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0.

0b0   This control does not cause any instructions to be trapped.

0b1   EL0 using AArch64: EL0 accesses to the AArch64 DCC registers are trapped.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.ERR. Reads and writes of this bit are indirect accesses to EDSCR.ERR.

The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

•   When OSLSR_EL1.OSLK == 1, access to this field is RW.

•   When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bits [5:1]

Reserved, RES0.

SS, bit [0]

Software step control bit. If EL_D is using AArch64, enable Software step. Permitted values are:

0b0   Software step disabled

0b1   Software step enabled.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the MDSCR_EL1

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.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, MDSCR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return MDSCR_EL1;
elsif PSTATE.EL == EL2 then
    return MDSCR_EL1;
```

MSR MDSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MDSCR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    MDSCR_EL1 = X[t];
```

G1.5 Performance Monitors registers

This section lists the Performance Monitoring registers in Armv8-R AArch64, in alphabetical order.

G1.5.1 PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configurations

AArch64 System register PMCCFILTR_EL0[31:0] is architecturally mapped to External register PMCCFILTR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are UNDEFINED.

Attributes

PMCCFILTR_EL0 is a 64-bit register.
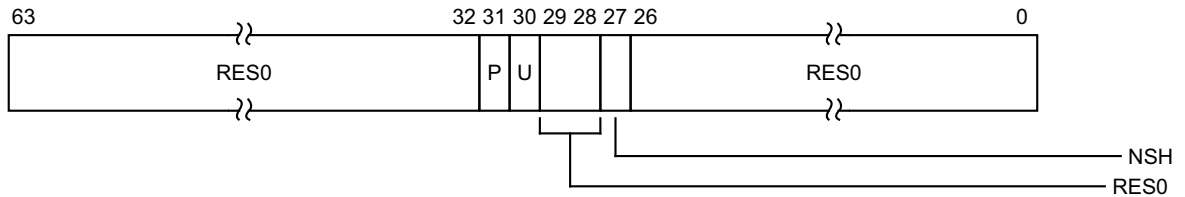
Field descriptions

The PMCCFILTR_EL0 bit assignments are:

Bits [63:32]

Reserved, RES0.

P, bit [31]

Privileged filtering bit. Controls counting in EL1.

0b0   Count cycles in EL1.

0b1   Do not count cycles in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

0b0   Count cycles in EL0.

0b1   Do not count cycles in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

0b0   Do not count cycles in EL2.

0b1   Count cycles in EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:0]

Reserved, RES0.

Accessing the PMCCFILTR_EL0

PMCCFILTR_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to 0b11111.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCCFILTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PMCCFILTR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PMCCFILTR_EL0;
elsif PSTATE.EL == EL2 then
    return PMCCFILTR_EL0;
```

MSR PMCCFILTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCCFILTR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCCFILTR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    PMCCFILTR_EL0 = X[t];
```

G1.5.2 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[7:0] is architecturally mapped to External register PMCR_EL0[7:0].

> This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR_EL0 are UNDEFINED.

Attributes

> PMCR_EL0 is a 64-bit register.

Field descriptions

The PMCR_EL0 bit assignments are:
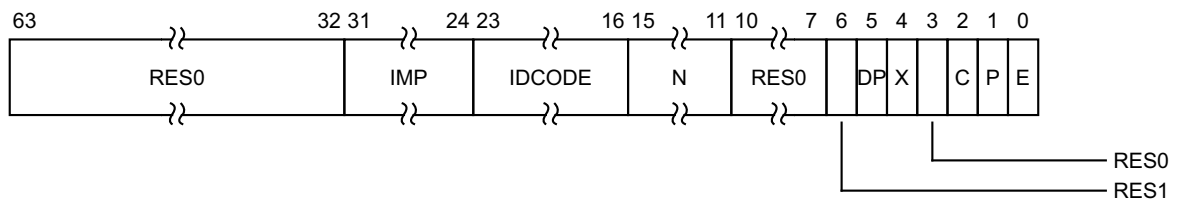
Bits [63:32]

> Reserved, RES0.

IMP, bits [31:24]

> Implementer code.

> If this field is zero, then PMCR_EL0.IDCODE is RES0 and software must use MIDR_EL1 to identify the PE.

> Otherwise, this field and PMCR_EL0.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A non-zero value has the same interpretation as MIDR_EL1.Implementer.

> Use of this field is deprecated.

> This field reads as an IMPLEMENTATION DEFINED value.

> Access to this field is RO.

IDCODE, bits [23:16]

> *When PMCR_EL0.IMP != 0x00:*

> > Identification code. Use of this field is deprecated. This field has an IMPLEMENTATION DEFINED value.

> > Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

> > Access to this field is RO.

> *Otherwise:*

> > Reserved, RES0.

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of `0b00000`-`0b11111`. If the value is `0b00000` then only PMCCNTR_EL0 is implemented. If the value is `0b11111` PMCCNTR_EL0 and 31 event counters are implemented.

When EL2 is implemented and enabled for the current Security state, reads of this field from EL1 and EL0 return the value of MDCR_EL2.HPMN.

Access to this field is RO.

Bits [10:7]

Reserved, RES0.

Bit [6]

Reserved, RES1.

DP, bit [5]

When FEAT_PMUv3p1 is implemented:

Disable cycle counter when event counting is prohibited.

`0b0`   Cycle counting by PMCCNTR_EL0 is not affected by this bit.

`0b1`   When event counting for counters in the range [0..(MDCR_EL2.HPMN-1)] is prohibited, cycle counting by PMCCNTR_EL0 is disabled.

For more information see 'Prohibiting event counting'.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

`0b0`   Do not export events.

`0b1`   Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL PE trace unit.

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.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [3]

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this bit are:

`0b0`   No action.

`0b1`   Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

────── **Note** ──────

Resetting PMCCNTR_EL0 does not change the cycle counter overflow bit.

The value of PMCR_EL0.LC is ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO.

P, bit [1]

Event counter reset. The effects of writing to this bit are:

0b0   No action.

0b1   Reset all event counters accessible in the current Exception level, not including
            PMCCNTR_EL0, to zero.

This bit is always RAZ.

In EL0 and EL1:

•   If EL2 is implemented and enabled in the current Security state, and MDCR_EL2.HPMN is
            less than PMCR_EL0.N, a write of 1 to this bit does not reset event counters in the range
            [MDCR_EL2.HPMN..(PMCR_EL0.N-1)].

•   If EL2 is not implemented, EL2 is disabled in the current Security state, or
            MDCR_EL2.HPMN equals PMCR_EL0.N, a write of 1 to this bit resets all the event
            counters.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

—————— **Note** ——————

Resetting the event counters does not change the event counter overflow bits.

————————————————————

Access to this field is WO.

E, bit [0]

Enable.

0b0   All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are disabled.

0b1   All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are enabled by
            PMCNTENSET_EL0.

If EL2 is implemented, then:

•   If EL2 is using AArch64, PMN is MDCR_EL2.HPMN.

•   If PMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in
            the range [PMN..(PMCR_EL0.N-1)].

If EL2 is not implemented, PMN is PMCR_EL0.N.

—————— **Note** ——————

The effect of MDCR_EL2.HPMN or HDCR.HPMN on the operation of this bit always applies if
EL2 is implemented, at all Exception levels including EL2 and EL3, and regardless of whether EL2
is enabled in the current Security state. For more information, see the description of
MDCR_EL2.HPMN or HDCR.HPMN.

————————————————————

On a Warm reset, this field resets to 0.

Accessing the PMCR_EL0

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMCR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PMCR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PMCR_EL0;
elsif PSTATE.EL == EL2 then
    return PMCR_EL0;
```

MSR PMCR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    PMCR_EL0 = X[t];
```

G1.5.3 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[31:0] is architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are UNDEFINED.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.
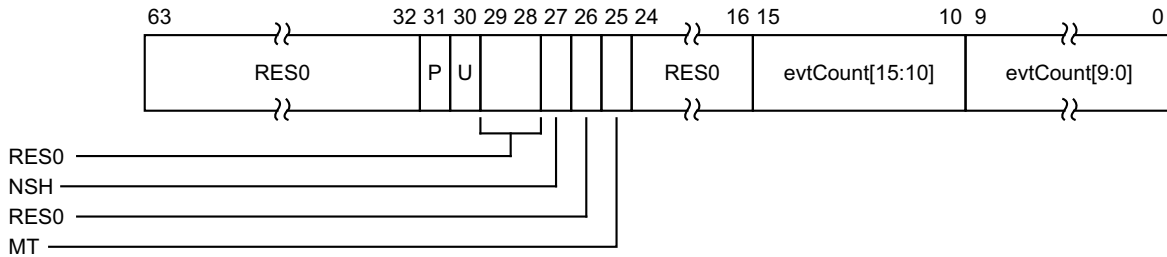
Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:

Bits [63:32]

Reserved, RES0.

P, bit [31]

Privileged filtering bit. Controls counting in EL1.

0b0   Count events in EL1.

0b1   Do not count events in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

0b0   Count events in EL0.

0b1   Do not count events in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

0b0   Do not count events in EL2.

0b1   Count events in EL2.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [26]

    Reserved, RES0.

MT, bit [25]

    When an IMPLEMENTATION DEFINED multi-threaded PMU Extension is implemented:

    Multithreading.

    0b0   Count events only on controlling PE.

    0b1   Count events from any PE with the same affinity at level 1 and above as this PE.

    ————— **Note** —————

- When the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach, an implementation is described as multi-threaded. That is, the performance of PEs at the lowest affinity level is highly interdependent.

- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

    On a Warm reset, this field resets to an architecturally UNKNOWN value.

    Otherwise:

    Reserved, RES0.

Bits [24:16]

    Reserved, RES0.

evtCount[15:10], bits [15:10]

    When FEAT_PMUv3p1 is implemented:

    Extension to evtCount[9:0]. See evtCount[9:0] for more details.

    On a Warm reset, this field resets to an architecturally UNKNOWN value.

    Otherwise:

    Reserved, RES0.

evtCount[9:0], bits [9:0]

    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.

    The ranges of event numbers allocated to each type of event are shown in 'Allocation of the PMU event number space'.

    If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range 0x0000 to 0x003F, 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.

- If 16-bit evtCount is implemented, for the range 0x4000 to 0x403F, 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.

    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.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVTYPER<n>_EL0

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 event counters, then 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.

- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

———— **Note** ————

In EL0, an access is permitted if it is enabled by PMUSERNR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. See MDCR_EL2.HPMN for more details.

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, PMEVTYPER<n>_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
elsif PSTATE.EL == EL2 then
    return PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)];
```

MSR PMEVTYPER<n>_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:n[4:3]	n[2:0]

```
if PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
```

```
                        AArch64.SystemAccessTrap(EL2, 0x18);
                    else
                        AArch64.SystemAccessTrap(EL1, 0x18);
            elsif MDCR_EL2.TPM == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
        elsif PSTATE.EL == EL1 then
            if MDCR_EL2.TPM == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
        elsif PSTATE.EL == EL2 then
            PMEVTYPER_EL0[UInt(CRm<1:0>:op2<2:0>)] = X[t];
```

Chapter G2
System Registers in a VMSA Implementation

This chapter describes the System registers in a VMSA implementation. It contains the following section:

- *General system control registers* on page G2-258.

G2.1   General system control registers

This section lists the System registers in a VMSA implementation of Armv8-R AArch64.

G2.1.1 TTBR1_EL1, Translation Table Base Register 1 (EL1)

The TTBR1_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL1&0 stage 1 translation regime, and other information for this translation regime.
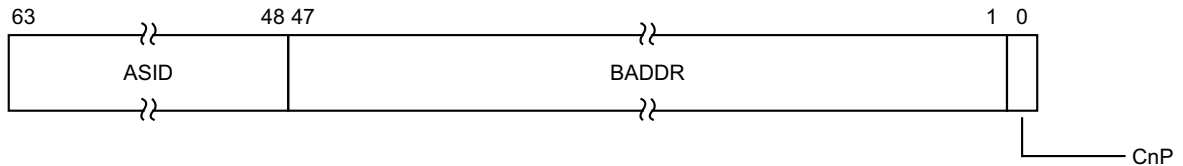
Configurations

In a PMSAv8-64 only implementation, this register is UNDEFINED.

Attributes

TTBR1_EL1 is a 64-bit register.

Field descriptions

The TTBR1_EL1 bit assignments are:

ASID, bits [63:48]

When VTCR_EL2.MSA == 1:

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.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BADDR, bits [47:1]

When VTCR_EL2.MSA == 1:

Translation table base address, A[47:x] or A[51:x], bits[47:1].

———— **Note** ————

•   Translation table base addresses of 52 bits, A[51:x], are supported only in an implementation that includes FEAT_LPA and is using the 64KB translation granule.

•   A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

————————————

In an implementation that includes FEAT_LPA, if the value of TCR_EL1.IPS is 0b110, then:

•   Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:

—   If x >= 6 then z=x.

—   Otherwise, z=6.

•   Register bits[5:2] hold bits[51:48] of the stage 1 translation table base address.

- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.

- When x>6 register bits[(x-1):6] are RES0.

- Register bit[1] is RES0.

- Bits[5:2] of the stage 1 translation table base address are zero.

- In an implementation that includes FEAT_TTCNP bit[0] of the stage 1 translation table base address is zero.

───── **Note** ─────

- In an implementation that includes FEAT_LPA a TCR_EL1.IPS value of 0b110, that selects an IPA size of 52 bits, is permitted only when using the 64KB translation granule.

- When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the Effective value of TCR_EL1.IPS is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

─────────────

If the Effective value of TCR_EL1.IPS is not 0b110 then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.

- Register bits[(x-1):1] are RES0.

- If the implementation supports 52-bit PAs and IPAs, then bits[51:48] of the translation table base addresses used in this stage of translation are 0b0000.

───── **Note** ─────

This definition applies:

- To an implementation that includes FEAT_LPA and is using a translation granule smaller than 64KB.

- To any implementation that does not include FEAT_LPA.

─────────────

If any TTBR1_EL1[47:0] bit that is defined as RES0 has the value 1 when a translation table walk is performed using TTBR1_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register 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.

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CnP, bit [0]

When VTCR_EL2.MSA == 1 and FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1.

0b0   The translation table entries pointed to by TTBR1_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR1_EL1 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR1_EL1.CnP on those other PEs.

- The value of the current ASID.

> • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

0b1   The translation table entries pointed to by TTBR1_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1 and all of the following apply:

> • The translation table entries are pointed to by TTBR1_EL1.

> • The translation tables relate to the same translation regime.

> • The ASID is the same as the current ASID.

> • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This field is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

———— **Note** ————

If the value of the TTBR1_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

————————————————

This field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the TTBR1_EL1

Accesses to this register use the following encodings in the System instruction encoding space:

MRS <Xt>, TTBR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    elsif HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '0' then
        UNDEFINED;
    else
        return TTBR1_EL1;
elsif PSTATE.EL == EL2 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    else
        return TTBR1_EL1;
```

MSR TTBR1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    elsif HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '0' then
        UNDEFINED;
    else
        TTBR1_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    else
        TTBR1_EL1 = X[t];
```

Part H

Armv8-R AArch64 External Debug Registers

Chapter H1
External Debug Registers Descriptions

This chapter provides the information on the external debug registers that are supported in Armv8-R AArch64. It contains the following sections:

H1.1   About the external debug registers

Armv8-R AArch64 supports both self-hosted and external debug as defined in the Armv8.4-A debug architecture, without the EL3 Exception level.

- Self-hosted debug: The PE hosts a debugger. The debugger programs the PE to generate debug exceptions. Debug exceptions are accommodated in the Armv8-R AArch64 Exception model.

- External debug: The PE is controlled by an external debugger. The debugger programs the PE to generate debug events that cause the PE to enter the debug state. In the debug state, the PE is halted.

For more information, see chapter *External System Control Register Descriptions of the Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

H1.2   External debug registers

This section describes the modified external debug registers for Armv8-R AArch64.

H1.2.1   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.

Configurations

External register DBGBCR<n>_EL1[31:0] is architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

DBGBCR<n>_EL1 is in the Core power domain.

If breakpoint n is not implemented then accesses to this register are:

- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().

- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

Attributes

DBGBCR<n>_EL1 is a 32-bit register.

Field descriptions

The DBGBCR<n>_EL1 bit assignments are:
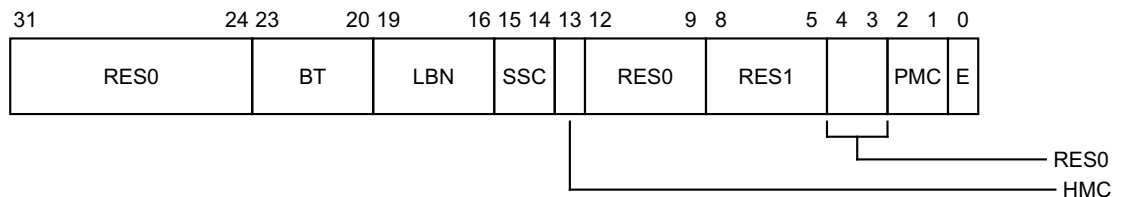
When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

0b0000   Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.

0b0001   As 0b0000 but linked to a Context matching breakpoint.

0b0010   Unlinked Context ID match. When FEAT_VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL2 value. Otherwise, DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL1 value.

0b0011   As 0b0010, with linking enabled.

0b0100   Unlinked instruction address mismatch. DBGBVR<n>_EL1 is the address of an instruction to be stepped.

0b0101   As 0b0100, with linking enabled.

0b0110   Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.

0b0111   As 0b0110, with linking enabled.

0b1000	Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VSCTLR_EL2.VMID.
0b1001	As 0b1000, with linking enabled.
0b1010	Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VSCTLR_EL2.VMID.
0b1011	As 0b1010, with linking enabled.
0b1100	Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.
0b1101	As 0b1100, with linking enabled.
0b1110	Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.
0b1111	As 0b1110, with linking enabled.

Constraints on breakpoint programming mean some values are reserved under certain conditions.

For more information on the operation of the SSC, HMC, and PMC fields, and on the effect of programming this field to a reserved value, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions' and 'Reserved DBGBCR<n>_EL1.BT values'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On an External debug reset, the value of this field is unchanged.

On a Warm reset, the value of this field is unchanged.

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.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On an External debug reset, the value of this field is unchanged.

On a Warm reset, the value of this field is unchanged.

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'.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On an External debug reset, the value of this field is unchanged.

On a Warm reset, the value of this field is unchanged.

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.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On an External debug reset, the value of this field is unchanged.

On a Warm reset, the value of this field is unchanged.

Bits [12:9]

Reserved, RES0.

Bits [8:5]

Reserved, RES1.

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.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On an External debug reset, the value of this field is unchanged.

On a Warm reset, the value of this field is unchanged.

E, bit [0]

Enable breakpoint DBGBVR<n>_EL1. Possible values are:

0b0   Breakpoint disabled.

0b1   Breakpoint enabled.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

On an External debug reset, the value of this field is unchanged.

On a Warm reset, the value of this field is unchanged.

Accessing the DBGBCR<n>_EL1:

——— **Note** ———

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGBCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x408 + (16 * n)	DBGBCR<n>_EL1

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.

- Otherwise accesses to this register generate an error response.

H1.2.2   EDAA32PFR, External Debug Auxiliary Processor Feature Register

The EDAA32PFR characteristics are:

Purpose

Provides information about implemented PE features.

— **Note** —

The register mnemonic, EDAA32PFR, is derived from previous definitions of this register that defined this register only when AArch64 was not supported at any Exception level.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configurations

It is IMPLEMENTATION DEFINED whether EDAA32PFR is implemented in the Core power domain or in the Debug power domain.

Attributes

EDAA32PFR is a 64-bit register.

Field descriptions

The EDAA32PFR bit assignments are:

63		20 19	16 15	12 11	8 7	4 3	0
RES0		MSA_frac	EL3	EL2	PMSA	VMSA	
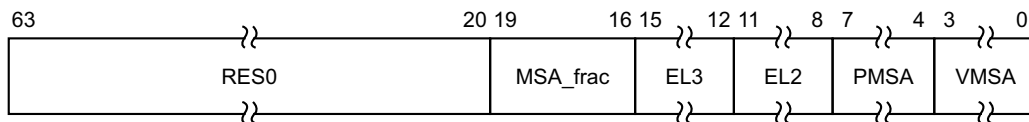
Bits [63:20]

Reserved, RES0.

MSA_frac, bits [19:16]

When EDAA32PFR.PMSA == 0b0000 and EDAA32PFR.VMSA == 0b1111:

Memory System Architecture fractional field. This holds the information on additional Memory System Architectures supported. Defined values are:

0b0001   PMSAv8-64 supported in all translation regimes. VMSAv8-64 not supported.

0b0010   PMSAv8-64 supported in all translation regimes. In addition to PMSAv8-64, stage 1 EL1&0 translation regime also supports VMSAv8-64.

All other values are reserved.

In Armv8-R AArch64, the only permitted values are 0b0001 and 0b0010.

Otherwise:

Reserved, RES0.

EL3, bits [15:12]

When EDPFR.EL3 == 0b0000:

AArch32 EL3 Exception level handling. Defined values are:

0b0000   EL3 is not implemented or can be executed in AArch64 state.

0b0001   EL3 can be executed in AArch32 state only.

All other values are reserved.

───── **Note** ─────

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

Otherwise:

Reserved, RAZ.

EL2, bits [11:8]

When EDPFR.EL2 == 0b0000:

AArch32 EL2 Exception level handling. Defined values are:

0b0000   EL2 is not implemented or can be executed in AArch64 state.

0b0001   EL2 can be executed in AArch32 state only.

All other values are reserved.

───── **Note** ─────

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

Otherwise:

Reserved, RAZ.

PMSA, bits [7:4]

Indicates support for a 32-bit PMSA. Defined values are:

0b0000   PMSA-32 not supported.

0b0100   PMSAv8-32 supported.

All other values are reserved.

VMSA, bits [3:0]

When EDAA32PFR.PMSA != 0b0000:

Indicates support for a VMSA in addition to a 32-bit PMSA Defined values are:

0b0000   VMSA not supported.

All other values are reserved.

When EDAA32PFR.PMSA == 0b0000:

Defined values are:

0b0000   VMSAv8-64 supported.
          PMSAv8-64 not supported.

0b1111   Memory system architecture described by EDAA32PFR.MSA_frac.

All other values are reserved.

In Armv8-R AArch64, the only permitted value is 0b1111.

Otherwise:

Reserved, RAZ.

Accessing the EDAA32PFR:

EDAA32PFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD60	EDAA32PFR

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to this register are RO.

- Otherwise accesses to this register are IMPDEF.

H1.2.3 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

Purpose

>   Provides information about implemented PE features.
>
>   For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configurations

>   It is IMPLEMENTATION DEFINED whether EDPFR is implemented in the Core power domain or in the Debug power domain.

Attributes

>   EDPFR is a 64-bit register.

Field descriptions

The EDPFR bit assignments are:
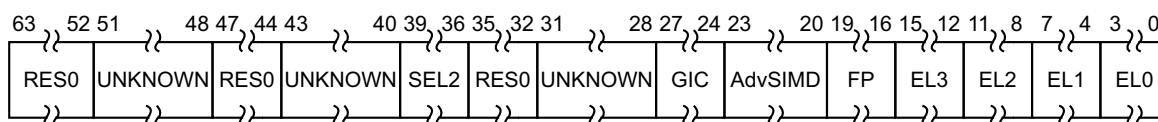
Bits [63:52]

>   Reserved, RES0.

UNKNOWN, bits [51:48]

>   *From Armv8.4:*
>
>   Reserved, UNKNOWN.
>
>   *Otherwise:*
>
>   Reserved, RES0.

Bits [47:44]

>   Reserved, RES0.

UNKNOWN, bits [43:40]

>   *From Armv8.2:*
>
>   Reserved, UNKNOWN.
>
>   *Otherwise:*
>
>   Reserved, RES0.

SEL2, bits [39:36]

>   Secure EL2. Defined values are:
>
>   0b0000   Secure EL2 is not implemented.
>
>   0b0001   Secure EL2 is implemented.
>
>   All other values are reserved.

Bits [35:32]

>   Reserved, RES0.

UNKNOWN, bits [31:28]

> ***From Armv8.2:***
>
> > Reserved, UNKNOWN.
>
> ***Otherwise:***
>
> > Reserved, RES0.

GIC, bits [27:24]

> System register GIC interface support. Defined values are:
>
> 0b0000   GIC CPU interface system registers not implemented.
>
> 0b0001   System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
>
> 0b0011   System register interface to version 4.1 of the GIC CPU interface is supported.
>
> All other values are reserved.
>
> In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.GIC.

AdvSIMD, bits [23:20]

> Advanced SIMD. Defined values are:
>
> 0b0000   Advanced SIMD is implemented, including support for the following SISD and SIMD operations:
>
> > - Integer byte, halfword, word and doubleword element operations.
> > - Single-precision and double-precision floating-point arithmetic.
> > - Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
>
> 0b0001   As for 0b0000, and also includes support for half-precision floating-point arithmetic.
>
> 0b1111   Advanced SIMD is not implemented.
>
> All other values are reserved.
>
> This field must have the same value as the FP field.
>
> The permitted values are:
>
> - 0b0000 in an implementation with Advanced SIMD support, that does not include the FEAT_FP16 extension.
> - 0b0001 in an implementation with Advanced SIMD support, that includes the FEAT_FP16 extension.
> - 0b1111 in an implementation without Advanced SIMD support.
>
> In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.AdvSIMD.

FP, bits [19:16]

> Floating-point. Defined values are:
>
> 0b0000   Floating-point is implemented, and includes support for:
>
> > - Single-precision and double-precision floating-point types.
> > - Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
>
> 0b0001   As for 0b0000, and also includes support for half-precision floating-point arithmetic.
>
> 0b1111   Floating-point is not implemented.
>
> All other values are reserved.
>
> This field must have the same value as the AdvSIMD field.

The permitted values are:

- `0b0000` in an implementation with floating-point support, that does not include the FEAT_FP16 extension.
- `0b0001` in an implementation with floating-point support, that includes the FEAT_FP16 extension.
- `0b1111` in an implementation without floating-point support.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.FP.

EL3, bits [15:12]

AArch64 EL3 Exception level handling. Defined values are:

`0b0000`   EL3 is not implemented or cannot be executed in AArch64 state.

`0b0001`   EL3 can be executed in AArch64 state only.

`0b0010`   EL3 can be executed in both Execution states.

When the value of EDAA32PFR.EL3 is non-zero, this field must be `0b0000`.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL3.

EL2, bits [11:8]

AArch64 EL2 Exception level handling. Defined values are:

`0b0000`   EL2 is not implemented or cannot be executed in AArch64 state.

`0b0001`   EL2 can be executed in AArch64 state only.

`0b0010`   EL2 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL2.

EL1, bits [7:4]

AArch64 EL1 Exception level handling. Defined values are:

`0b0000`   EL1 cannot be executed in AArch64 state.

`0b0001`   EL1 can be executed in AArch64 state only.

`0b0010`   EL1 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL1.

EL0, bits [3:0]

AArch64 EL0 Exception level handling. Defined values are:

`0b0000`   EL0 cannot be executed in AArch64 state.

`0b0001`   EL0 can be executed in AArch64 state only.

`0b0010`   EL0 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64 state in at least one Exception level, this field returns the value of ID_AA64PFR0_EL1.EL0.

Accessing the EDPFR:

EDPFR[31:0] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	0xD20	EDPFR	31:0

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDPFR[31:0] are RO.

- Otherwise accesses to EDPFR[31:0] are IMPDEF.

EDPFR[63:32] can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	0xD24	EDPFR	63:32

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDPFR[63:32] are RO.

- Otherwise accesses to EDPFR[63:32] are IMPDEF.

H1.2.4   PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configurations

External register PMCCFILTR_EL0[31:0] is architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

PMCCFILTR_EL0 is in the Core power domain.

On a Warm or Cold reset, RW fields in this register reset:

- To architecturally UNKNOWN values if the reset is to an Exception level that is using AArch64.

- To 0 if the reset is to an Exception level that is using AArch32.

The register is not affected by an External debug reset.

Attributes

PMCCFILTR_EL0 is a 32-bit register.
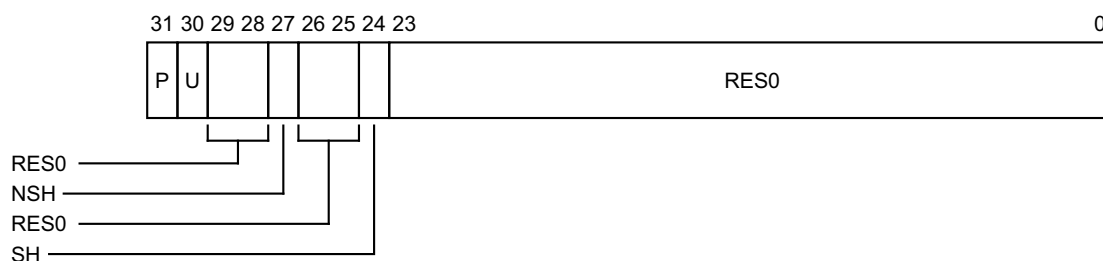
Field descriptions

The PMCCFILTR_EL0 bit assignments are:

P, bit [31]

Privileged filtering bit. Controls counting in EL1.

0b0	Count cycles in EL1.
0b1	Do not count cycles in EL1.

U, bit [30]

User filtering bit. Controls counting in EL0.

0b0	Count cycles in EL0.
0b1	Do not count cycles in EL0.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

> EL2 (Hypervisor) filtering bit. Controls counting in EL2.
>
> 0b0   Do not count cycles in EL2.
>
> 0b1   Count cycles in EL2.

Bits [26:25]

> Reserved, RES0.

SH, bit [24]

> *When ARMv8.4-SecEL2 is implemented:*
>
> Secure EL2 filtering.
>
> If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Secure EL2 are counted.
>
> Otherwise, cycles in Secure EL2 are not counted.
>
> If Secure EL2 is disabled, this field is RES0.
>
> *Otherwise:*
>
> Reserved, RES0.

Bits [23:0]

> Reserved, RES0.

Accessing the PMCCFILTR_EL0:

—— **Note** ——

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCCFILTR_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	0x47C	PMCCFILTR_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.

- Otherwise accesses to this register generate an error response.

H1.2.5 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

External register PMCR_EL0[7:0] is architecturally mapped to AArch64 System register PMCR_EL0[7:0].

PMCR_EL0 is in the Core power domain.

Attributes

PMCR_EL0 is a 32-bit register.

Field descriptions

The PMCR_EL0 bit assignments are:
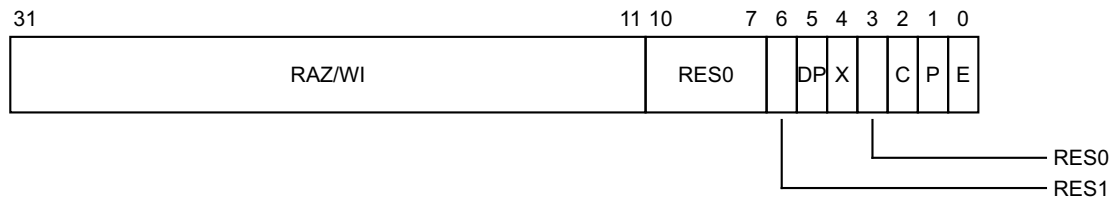
Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bits [10:7]

Reserved, RES0.

Bit [6]

Reserved, RES1.

DP, bit [5]

When ARMv8.1-PMU is implemented :

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

0b0   Cycle counting by PMCCNTR_EL0 is not affected by this bit.

0b1   When event counting for counters in the range [0..(MDCR_EL2.HPMN-1)] is prohibited, cycle counting by PMCCNTR_EL0 is disabled.

For more information, see 'Prohibiting event counting'.

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.

Otherwise:

    Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

    Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

    0b0   Do not export events.

    0b1   Export events where not prohibited.

    This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL PE trace unit.

    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.

Otherwise:

    Reserved, RAZ/WI.

Bit [3]

    Reserved, RES0.

C, bit [2]

    Cycle counter reset. The effects of writing to this bit are:

    0b0   No action.

    0b1   Reset PMCCNTR_EL0 to zero.

    This bit is always RAZ.

    ———— **Note** ————

    Resetting PMCCNTR_EL0 does not change the cycle counter overflow bit.

    Access to this field is WO.

P, bit [1]

    Event counter reset. The effects of writing to this bit are:

    0b0   No action.

    0b1   Reset all event counters, not including PMCCNTR_EL0, to zero.

    This bit is always RAZ.

    ———— **Note** ————

    Resetting the event counters does not change the event counter overflow bits.

    Access to this field is WO.

E, bit [0]

Enable.

0b0   All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are disabled.

0b1   All event counters in the range [0..(PMN-1)] and PMCCNTR_EL0, are enabled by
        PMCNTENSET_EL0.

If EL2 is implemented then:

•   If EL2 is using AArch64, PMN is MDCR_EL2.HPMN.

•   If PMN is less than PMCR_EL0.N, this bit does not affect the operation of event counters in
        the range [PMN..(PMCR_EL0.N-1)].

If EL2 is not implemented, PMN is PMCR_EL0.N.

——— **Note** ———

The effect of the following fields on the operation of this bit applies if EL2 is implemented
regardless of whether EL2 is enabled in the current Security state:

•   MDCR_EL2.HPMN. See the description of MDCR_EL2.HPMN for more information.

On a Warm reset, this field resets to 0.

Accessing the PMCR_EL0:

——— **Note** ———

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new
definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCR_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	0xE04	PMCR_EL0

This interface is accessible as follows:

•   When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and
        SoftwareLockStatus() accesses to this register are RO.

•   When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and
        !SoftwareLockStatus() accesses to this register are RW.

•   Otherwise accesses to this register generate an error response.

H1.2.6 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

External register PMEVTYPER<n>_EL0[31:0] is architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[31:0].

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented then accesses to this register are:

- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess().

- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

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.

0b0   Count events in EL1.

0b1   Do not count events in EL1.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

0b0   Count events in EL0.

0b1   Do not count events in EL0.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

> EL2 (Hypervisor) filtering bit. Controls counting in EL2.
>
> 0b0   Do not count events in EL2.
>
> 0b1   Count events in EL2.
>
> On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

> Reserved, RES0.

Bit [26]

> Reserved, RES0.

MT, bit [25]

> ***When an IMPLEMENTATION DEFINED multi-threaded PMU Extension is implemented:***
>
> Multithreading.
>
> 0b0   Count events only on controlling PE.
>
> 0b1   Count events from any PE with the same affinity at level 1 and above as this PE.

> ———— **Note** ————
>
> - When the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach, an implementation is described as multi-threaded. That is, the performance of PEs at the lowest affinity level is highly interdependent.
>
> - Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

> On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

> Reserved, RES0.

SH, bit [24]

> ***When ARMv8.4-SecEL2 is implemented:***
>
> Secure EL2 filtering.
>
> If the value of this bit is not equal to the value of the PMEVTYPER<n>_EL0.NSH bit, events in Secure EL2 are counted.
>
> Otherwise, events in Secure EL2 are not counted.
>
> On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

> Reserved, RES0.

Bits [23:16]

> Reserved, RES0.

evtCount[15:10], bits [15:10]

> ***When ARMv8.1-PMU is implemented:***
>
> Extension to evtCount[9:0]. See evtCount[9:0] for more details.
>
> On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

> Reserved, RES0.

evtCount[9:0], bits [9:0]

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.

The ranges of event numbers allocated to each type of event are shown in 'Allocation of the PMU
event number space'.

If evtCount is programmed to an event that is reserved or not supported by the PE, the behavior
depends on the value written:

- For the range 0x0000 to 0x003F, 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.

- If 16-bit evtCount is implemented, for the range 0x4000 to 0x403F, 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.

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.

On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMEVTYPER<n>_EL0:

───── **Note** ─────

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new
definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMEVTYPER<n>_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
PMU	0x400 + 4n	PMEVTYPER<n>_EL0

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and
  SoftwareLockStatus() accesses to this register are RO.

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and
  !SoftwareLockStatus() accesses to this register are RW.

- Otherwise accesses to this register generate an error response.

Part I
Architectural Pseudocode

Chapter I1
Armv8-R AArch64 Pseudocode

This chapter contains the pseudocode that describes many features of the Armv8-R AArch64 architecture. It contains the following sections:

I1.1 Pseudocode for AArch64 operations

This section provides the architectural pseudocode for execution in AArch64 state. Functions that are listed in this section are identified as AArch64.FunctionName.

This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example aarch64/debug/breakpoint.

The top-level sections of the AArch64 pseudocode hierarchy are:

- *aarch64/debug*.
- *aarch64/exceptions* on page I1-297.
- *aarch64/functions* on page I1-312.
- *aarch64/instrs* on page I1-343.
- *aarch64/translation* on page I1-350.

I1.1.1 aarch64/debug

This section includes the following pseudocode functions:

- *aarch64/debug/breakpoint/AArch64.BreakpointMatch*.
- *aarch64/debug/breakpoint/AArch64.BreakpointValueMatch* on page I1-291.
- *aarch64/debug/breakpoint/AArch64.StateMatch* on page I1-292.
- *aarch64/debug/enables/AArch64.GenerateDebugExceptions* on page I1-293.
- *aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom* on page I1-293.
- *aarch64/debug/pmu/AArch64.CheckForPMUOverflow* on page I1-293.
- *aarch64/debug/pmu/AArch64.CountEvents* on page I1-294.
- *aarch64/debug/statisticalprofiling/TimeStamp* on page I1-295.
- *aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState* on page I1-295.
- *aarch64/debug/watchpoint/AArch64.WatchpointByteMatch* on page I1-295.
- *aarch64/debug/watchpoint/AArch64.WatchpointMatch* on page I1-296.

aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then                // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();
```

```
        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.
    dbgtype = DBGBCR_EL1[n].BT;

    if ((dbgtype IN {'011x','11xx'} && !HaveV82Debug()) ||          // Context matching
         dbgtype == '010x' ||                                       // Reserved
        (dbgtype != '0x0x' && !context_aware) ||                    // Context matching
        (dbgtype == '1xxx' && !HaveEL(EL2))) then                   // EL2 extension
        (c, dbgtype) = 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 = (dbgtype == '0x0x');
    match_vmid = (dbgtype == '10xx');
    match_cid1 = (dbgtype == 'xx1x');
    match_cid2 = (dbgtype == '11xx');
    linked     = (dbgtype == 'xxx1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, 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>);
        assert byte == 0;                      // "vaddress" is word aligned
        byte_select_match = TRUE;              // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress, TRUE, PSTATE.EL);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid1 then
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    if match_vmid then
        if !Have16bitVMID() || VTCR_EL2.VS == '0' then
```

```
                     vmid = ZeroExtend(VSCTLR_EL2.VMID<7:0>, 16);
                     bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
                 else
                     vmid = VSCTLR_EL2.VMID;
                     bvr_vmid = DBGBVR_EL1[n]<47:32>;
                 BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                              vmid == bvr_vmid);
         elsif match_cid2 then
             BXVR_match = (HaveV82Debug() && EL2Enabled() &&
                          DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

         bvr_match_valid = (match_addr || match_cid1);
         bxvr_match_valid = (match_vmid || match_cid2);

         match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

         return match;
```

aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt,  boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    (c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    if !ispriv && !isbreakpnt then
        priv_match = EL0_match;
    else
        case PSTATE.EL of
            when EL3  priv_match = EL3_match;
            when EL2  priv_match = EL2_match;
            when EL1  priv_match = EL1_match;
            when EL0  priv_match = EL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;                  // Both
        when '01'  security_state_match = !IsSecure();           // Non-secure only
        when '10'  security_state_match = IsSecure();            // Secure only
        when '11'  security_state_match = (HMC == '1' || IsSecure()); // HMC=1 -> Both, 0 -> Secure
only

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (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 || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' ||
MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);
    enabled = TRUE;

    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);

    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_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
```

```
            // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

            return pmuirq;
```

aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)
    assert n == 31 || n < UInt(PMCR_EL0.N);

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        E = if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME;
    else
        E = PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    prohibited = FALSE;

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * Executing at EL2
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && HaveEL(EL2) && HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n
== 31) then
        prohibited = (MDCR_EL2.HPMD == '1');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if ExternalSecureNoninvasiveDebugEnabled() then
        prohibited = FALSE;

    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
    filter = if n == 31 then PMCCFILTR_EL0[31:0] else PMEVTYPER_EL0[n]<31:0>;

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M = if HaveEL(EL3) then filter<26> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (NSH == '0');
        when EL3 filtered = (M != P);

    return !debug && enabled && !prohibited && !filtered;
```

aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp {
    TimeStamp_None,            // No timestamp
    TimeStamp_CoreSight,       // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,        // Physical counter value with no offset
    TimeStamp_Virtual };       // Physical counter value minus CNTVOFF_EL2
```

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);

    sync_errors = HaveIESB() && SCTLR[target_el].IESB == '1';
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool() then
        sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(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;
    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 && SCTLR_EL1.SPAN == '0' then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                          // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// ============================

boolean AArch64.WatchpointByteMatch(integer n,  bits(64) vaddress)
```

```
                    el = PSTATE.EL;
                    top = AddrTop(vaddress, FALSE, 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;
```

I1.1.2 aarch64/exceptions

This section includes the following pseudocode functions:

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
                           AArch64.BreakpointException(fault);
                       else
                           AArch64.WatchpointException(vaddress, fault);
                   elsif 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 exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_Watchpoint};

    exception.syndrome = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = 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 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR_EL2.TGE == '1' ||
                        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                        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);
    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.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                        (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                         (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

    bits(64) preferred_exception_return = ThisInstrAddr();

    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

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 EL2Enabled() && 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 EL2Enabled() && 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 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==============================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// ==================================

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    route_to_el3 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    ClearPendingPhysicalSError();
```

```
        if PSTATE.EL == EL3 || route_to_el3 then
            AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
        elsif PSTATE.EL == EL2 || route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// ===============================

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// ===============================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ==================================

AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    if HaveRASExt() then
        exception.syndrome<24>   = VSESR_EL2.IDS;
        exception.syndrome<23:0> = VSESR_EL2.ISS;
    else
        exception.syndrome<24> = if impdef_syndrome then '1' else '0';
        if impdef_syndrome then exception.syndrome<23:0> = syndrome;

    ClearPendingVirtualSError();
    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 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ===========================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ==============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
```

```
        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 EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    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()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, 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 exceptype of
        when Exception_Uncategorized      ec = 0x00; il = '1';
        when Exception_WFxTrap            ec = 0x01;
        when Exception_CP15RTTrap         ec = 0x03;           assert from_32;
        when Exception_CP15RRTTrap        ec = 0x04;           assert from_32;
        when Exception_CP14RTTrap         ec = 0x05;           assert from_32;
        when Exception_CP14DTTrap         ec = 0x06;           assert from_32;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap           ec = 0x08;
```

```
        when Exception_PACTrap            ec = 0x09;
        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_PACFail            ec = 0x1C;
        when Exception_InstructionAbort   ec = 0x20; il = '1';
        when Exception_PCAlignment        ec = 0x22; il = '1';
        when Exception_DataAbort          ec = 0x24;
        when Exception_SPAlignment        ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException ec = 0x28;
        when Exception_SError             ec = 0x2F; il = '1';
        when Exception_Breakpoint         ec = 0x30; il = '1';
        when Exception_SoftwareStep       ec = 0x32; il = '1';
        when Exception_Watchpoint         ec = 0x34; il = '1';
        when Exception_SoftwareBreakpoint ec = 0x38;
        when Exception_VectorCatch        ec = 0x3A; il = '1'; assert from_32;
        otherwise                         Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);
```

aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
            if IsSecureEL2Enabled() && IsSecure() then
                HPFAR_EL2.NS = exception.NS;
            else
                HPFAR_EL2.NS = '0';
        else
            HPFAR_EL2<43:4> = bits(40) 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.DIT = '0';               // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';                // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                    // IMPLEMENTATION DEFINED reset vector

    rv = RVBAR_EL2;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType_RESET);
```

aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1';                    // TFV
        else
            exception.syndrome<23> = '0';                    // TFV
    else
        exception.syndrome<23> = '1';                        // TFV
```

```
        exception.syndrome<10:8> = bits(3) UNKNOWN;                // VECITR
        if exception.syndrome<23> == '1' then
            exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
        else
            exception.syndrome<7,4:0> = bits(6) UNKNOWN;

        route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    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);
    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)

    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
```

```
                bits(64) preferred_exception_return = NextInstrAddr();
                vect_offset = 0x0;

                exception = ExceptionSyndrome(Exception_SupervisorCall);
                exception.syndrome<15:0> = immediate;

                if UInt(PSTATE.EL) > UInt(EL1) then
                    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
                elsif route_to_el2 then
                    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
                else
                    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/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);

    sync_errors = HaveIESB() && SCTLR[target_el].IESB == '1';
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    spsr = GetPSRFromPSTATE();

    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = spsr;
    ELR[] = preferred_exception_return;

    PSTATE.SS = '0';
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then                               // Coming from AArch32
        PSTATE.IT = '00000000';
```

```
              PSTATE.T = '0';                          // PSTATE.J is RES0
        if HavePANExt() && PSTATE.EL == EL1 && SCTLR_EL1.SPAN == '0' then
            PSTATE.PAN = '1';
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;

        BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

        EndOfInstruction();
```

aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

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 PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;

        // 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_EL1.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

        AArch64.CheckFPAdvSIMDTrap();                  // Also check against CPTR_EL2 and CPTR_EL3
```

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ===========================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    return;
```

aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ===============================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// ========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';

    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 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// ========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0                                        // Trapped access due to unknown
reason.
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                                        // Trapped access to SVE, Advance
SIMD&FP system register.
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18                                       // Trapped access to system
register or system instruction.
```

```
                exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
                instr = ThisInstr();
                exception.syndrome<21:20> = instr<20:19>;        // Op0
                exception.syndrome<19:17> = instr<7:5>;          // Op2
                exception.syndrome<16:14> = instr<18:16>;        // Op1
                exception.syndrome<13:10> = instr<15:12>;        // CRn
                exception.syndrome<9:5>  = instr<4:0>;           // Rt
                exception.syndrome<4:1>  = instr<11:8>;          // CRm
                exception.syndrome<0>    = instr<21>;            // Direction
            otherwise
                Unreachable();

        return exception;
```

aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// =======================

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

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 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

I1.1.3 aarch64/functions

This section includes the following pseudocode functions:

aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// ==========================

FaultRecord AArch64.CreateFaultRecord(Fault statuscode, bits(52) ipaddress, boolean NS,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(2) errortype, boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.statuscode = statuscode;
    fault.domain = bits(4) UNKNOWN;         // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;       // Not used from AArch64
    fault.errortype = errortype;
```

```
        fault.ipaddress.NS = if NS then '1' else '0';
        fault.ipaddress.address = ipaddress;
        fault.level = level;
        fault.acctype = acctype;
        fault.write = write;
        fault.extflag = extflag;
        fault.secondstage = secondstage;
        fault.s2fs1walk = s2fs1walk;

        return fault;
```

aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// ======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, 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.statuscode, fault.level);

    return iss;
```

aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ==============================

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
```

```
        ClearExclusiveLocal(ProcessorID());

    if passed then
        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 Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.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 {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);

    FPRounding 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 && zero2) || (zero1 && inf2) then
                        result = FPOnePointFive('0');
                    elsif inf1 || inf2 then
                        result = FPInfinity(sign1 EOR sign2);
                    else
                        // Fully fused multiply-add and halve
                        result_value = (3.0 + (value1 * value2)) / 2.0;
                        if result_value == 0.0 then
                            // Sign of exact zero result depends on rounding mode
                            sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                            result = FPZero(sign);
                        else
                            result = FPRound(result_value, FPCR, rounding);

                return result;
```

aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);

    FPRounding 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 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR, rounding);
    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, AccType_ORDEREDATOMIC,
AccType_ORDEREDATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_ORDEREDATOMIC,
AccType_ORDEREDATOMICRW };
    vector  = acctype == AccType_VEC;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        check = (UInt(address<0+:4>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;

    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;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    value = _Mem[memaddrdesc, size, accdesc];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
```

```
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    _Mem[memaddrdesc, size, accdesc] = value;
    return;
```

aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```
// AArch64.TranslateAddressForAtomicAccess()
// =======================================
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    return memaddrdesc;
```

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/IsBlockDescriptorNTBitValid

```
// If the implementation supports changing the block size without a break-before-make
// approach, then for implementations that have level 1 or 2 support, the nT bit in
// the block descriptor is valid.
boolean IsBlockDescriptorNTBitValid();
```

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;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        value<63:0>   = AArch64.MemSingle[address,   8, acctype, aligned];
        value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
```

```
                // not, so we must be changing to a new translation page.
                if !aligned then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FAULT, Constraint_NONE};
                    if c == Constraint_NONE then aligned = TRUE;

                for i = 1 to size-1
                    AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
        elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
            AArch64.MemSingle[address,   8, acctype, aligned] = value<63:0>;
            AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
        else
            AArch64.MemSingle[address, size, acctype, aligned] = value;
        return;
```

aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType
stacctype)
    bits(size) newvalue;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc];
    if BigEndian() then
        oldvalue = BigEndianReverse(oldvalue);

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

    if BigEndian() then
        newvalue = BigEndianReverse(newvalue);
    _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;

    // Load operations return the old (pre-operation) value
    return oldvalue;
```

aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                   bits(size) newvalue, AccType ldacctype, AccType stacctype)
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);
```

```
                        // All observers in the shareability domain observe the
                        // following load and store atomically.
                        oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc];
                        if BigEndian() then
                            oldvalue = BigEndianReverse(oldvalue);

                        if oldvalue == expectedvalue then
                            if BigEndian() then
                                newvalue = BigEndianReverse(newvalue);
                            _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
                        return oldvalue;
```

aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxx1 with TBI1=0 and TBI0=1:
    // This include EL1/EL0 in both VMSA and PMSA context.
    if PSTATE.EL == EL1 || PSTATE.EL == EL0 then
        assert S1TranslationRegime() == EL1;
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
```

```
                    PAC = 0x0000000000000000<63:0>;
                elsif !HaveEnhancedPAC2() then
                    PAC<top_bit-1> = NOT(PAC<top_bit-1>);

        // preserve the determination between upper and lower address at bit<55> and insert PAC
        if !HaveEnhancedPAC2() then
            if tbi then
                result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
            else
                result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            if tbi then
                result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR
PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
            else
                result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR
                        PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
        return result;
```

aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1 ;
        Enable = SCTLR_EL1.EnDA;
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnDA;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else return AddPAC(X, Y, APDAKey_EL1, TRUE);
```

aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
```

```
        assert S1TranslationRegime() == EL1;
        Enable = SCTLR_EL1.EnDB;
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnDB;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else return AddPAC(X, Y, APDBKey_EL1, TRUE);
```

aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        TrapEL2 = FALSE;
    else
        Unreachable();

    if TrapEL2 then TrapPACUse(EL2);
    else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:Zeros(32);
```

aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1;
        Enable =  SCTLR_EL1.EnIA;
        TrapEL2 = HCR_EL2.API == '0';
    elsif  PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnIA;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else return AddPAC(X, Y, APIAKey_EL1, FALSE);
```

aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1;
        Enable = SCTLR_EL1.EnIB;
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnIB;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else return AddPAC(X, Y, APIBKey_EL1, FALSE);
```

aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PACFail);
    exception.syndrome<1:0>  = syndrome;
    exception.syndrome<24:2> = Zeros();                  // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

aarch64/functions/pac/auth/Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number, boolean
is_combined)
    bits(64) PAC;
    bits(64) result;
```

```
            bits(64) original_ptr;
            bits(2)  error_code;
            bits(64) extfield;

            // Reconstruct the extension field used of adding the PAC to the pointer
            boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
            integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
            extfield = Replicate(ptr<55>, 64);

            if tbi then
                original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
            else
                original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

            PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
            // Check pointer authentication code
            if tbi then
                if !HaveEnhancedPAC2() then
                    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                        result = original_ptr;
                    else
                        error_code = key_number:NOT(key_number);
                        result = original_ptr<63:55>:error_code:original_ptr<52:0>;
                else
                    result = ptr;
                    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
                    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                        if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                            error_code = (if data then '1' else '0'):key_number;
                            AArch64.PACFailException(error_code);
            else
                if !HaveEnhancedPAC2() then
                    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                        result = original_ptr;
                    else
                        error_code = key_number:NOT(key_number);
                        result = original_ptr<63>:error_code:original_ptr<60:0>;
                else
                    result = ptr;
                    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
                    result<63:56> = result<63:56> EOR PAC<63:56>;
                    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                        if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                            error_code = (if data then '1' else '0'):key_number;
                            AArch64.PACFailException(error_code);
            return result;
```

aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1;
        Enable = SCTLR_EL1.EnDA;
        TrapEL2 = HCR_EL2.API == '0';
```

```
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnDA;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else  return Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);
```

aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1;
        Enable =  SCTLR_EL1.EnDB;
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnDB;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else  return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);
```

aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1;
        Enable = SCTLR_EL1.EnIA;
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnIA;
        TrapEL2 = FALSE;
    else
        Unreachable();
```

```
    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else  return Auth(X, Y, APIAKey_EL1, FALSE, '0', is_combined);
```

aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if PSTATE.EL IN {EL0, EL1} then
        assert S1TranslationRegime() == EL1;
        Enable = SCTLR_EL1.EnIB;
        TrapEL2 = HCR_EL2.API == '0';
    elsif PSTATE.EL == EL2 then
        Enable = SCTLR_EL2.EnIB;
        TrapEL2 = FALSE;
    else
        Unreachable();

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    else  return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
```

aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() == EL1;
        tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
        using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';

        max_limit_tsz_field = (if !HaveSmallPageTblExt() then 39 else if using64k then 47 else 48);
        if tsz_field > max_limit_tsz_field then
            // TCR_ELx.TySZ is out of range
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_NONE};
            if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
        tszmin = if using64k && VAMax() == 52 then 12 else 16;
        if tsz_field < tszmin then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_NONE};
            if c == Constraint_FORCE then tsz_field = tszmin;
        return (64-tsz_field);
    else
        // For EL2 and  EL1 with PMSA context.
        return PAMax();
```

aarch64/functions/pac/computepac/ComputePAC

```
array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to 4
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to 4
        workingval = PACInvSub(workingval);
        if i < 4 then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[4-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
```

```
        outdata<15:12> = indata<39:36>;
        outdata<19:16> = indata<59:56>;
        outdata<23:20> = indata<47:44>;
        outdata<27:24> = indata<7:4>;
        outdata<31:28> = indata<19:16>;
        outdata<35:32> = indata<35:32>;
        outdata<39:36> = indata<55:52>;
        outdata<43:40> = indata<31:28>;
        outdata<47:44> = indata<11:8>;
        outdata<51:48> = indata<23:20>;
        outdata<55:52> = indata<3:0>;
        outdata<59:56> = indata<43:40>;
        outdata<63:60> = indata<63:60>;
        return outdata;
```

aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0>   = indata<55:52>;
    outdata<7:4>   = indata<27:24>;
    outdata<11:8>  = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher

    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0010';
            when '1001'  Toutput<4*i+3:4*i> = '0110';
            when '1010'  Toutput<4*i+3:4*i> = '1111';
            when '1011'  Toutput<4*i+3:4*i> = '0000';
            when '1100'  Toutput<4*i+3:4*i> = '0100';
            when '1101'  Toutput<4*i+3:4*i> = '1100';
```

```
                   when '1110'  Toutput<4*i+3:4*i> = '0111';
                   when '1111'  Toutput<4*i+3:4*i> = '0011';
          return Toutput;
```

aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '1011';
            when '0001'  Toutput<4*i+3:4*i> = '0110';
            when '0010'  Toutput<4*i+3:4*i> = '1000';
            when '0011'  Toutput<4*i+3:4*i> = '1111';
            when '0100'  Toutput<4*i+3:4*i> = '1100';
            when '0101'  Toutput<4*i+3:4*i> = '0000';
            when '0110'  Toutput<4*i+3:4*i> = '1001';
            when '0111'  Toutput<4*i+3:4*i> = '1110';
            when '1000'  Toutput<4*i+3:4*i> = '0011';
            when '1001'  Toutput<4*i+3:4*i> = '0111';
            when '1010'  Toutput<4*i+3:4*i> = '0100';
            when '1011'  Toutput<4*i+3:4*i> = '0101';
            when '1100'  Toutput<4*i+3:4*i> = '1101';
            when '1101'  Toutput<4*i+3:4*i> = '0010';
            when '1110'  Toutput<4*i+3:4*i> = '0001';
            when '1111'  Toutput<4*i+3:4*i> = '1010';
        return Toutput;
```

aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
```

```
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4)incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64)indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
```

```
        outdata<3:0> = indata<19:16>;
        outdata<7:4> = indata<23:20>;
        outdata<11:8> = TweakCellRot(indata<27:24>);
        outdata<15:12> = indata<31:28>;
        outdata<19:16> = TweakCellRot(indata<47:44>);
        outdata<23:20> = indata<11:8>;
        outdata<27:24> = indata<15:12>;
        outdata<31:28> = TweakCellRot(indata<35:32>);
        outdata<35:32> = indata<51:48>;
        outdata<39:36> = indata<55:52>;
        outdata<43:40> = indata<59:56>;
        outdata<47:44> = TweakCellRot(indata<63:60>);
        outdata<51:48> = TweakCellRot(indata<3:0>);
        outdata<55:52> = indata<7:4>;
        outdata<59:56> = TweakCellRot(indata<43:40>);
        outdata<63:60> = TweakCellRot(indata<39:36>);
        return outdata;
```

aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// ================
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return ( HavePACExt()
        && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
```

aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// =================
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC 2 functionality";
```

aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// =========
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// =================
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// ===========
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    return IsStage1VMSA() && (PSTATE.EL == EL1 || PSTATE.EL == EL0);
```

aarch64/functions/pac/strip/Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = FALSE;
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
```

```
        else
            mask_active = PSTATE.EL == target;

        mask_set = PSTATE.A == '1';
        intdis = Halted() || ExternalDebugInterruptsDisabled(target);
        masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

        // Check for a masked Physical SError pending that can be synchronized
        // by an Error synchronization event.
        if masked && IsSynchronizablePhysicalSErrorPending() then
            implicit_esb = FALSE;
            syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome)<31:0>;
            ClearPendingPhysicalSError();                // Set ISR_EL1.A to 0

        return;
```

aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// =============================
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31>   = '1';              // A
    target<24>   = syndrome<24>;     // IDS
    target<23:0> = syndrome<23:0>;   // ISS
    return target;
```

aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>)<31:0>;
        HCR_EL2.VSE = '0';                          // Clear pending virtual SError

    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;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool() 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);
    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>;
```

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 a value held in the register;
//   part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//   value held in the register.

bits(width) Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width IN {32,64};
        return _V[n]<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
//   part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//   part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<(width * 2)-1:width> = value<width-1:0>;
```

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[]
    bits(32) r;
    r = CNTKCTL_EL1;
    return r;
```

aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// ============================

CPACRType CPACR[]
    bits(32) r;
    r = CPACR_EL1;
    return r;
```

aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ==========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ==========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// ======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        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;
        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;
        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;
        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;
        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;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ===========================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

aarch64/functions/sysregisters/MPUIR

```
// MPUIR[] - non-assignment form
// ===========================

MPUIRType MPUIR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = MPUIR_EL1;
        when EL2  r = MPUIR_EL2;
        otherwise Unreachable();
    return r;
```

aarch64/functions/sysregisters/MPUIRType

```
type MPUIRType;
```

aarch64/functions/sysregisters/PRBARn

```
// PRBARn[] - non-assignment form
// ===========================
```

```
PRBARnType PRBARn[bits(2) regime, integer index]
    bits(64) r;
    case regime of
        when EL1  r = PRBARn_EL1[index];
        when EL2  r = PRBARn_EL2[index];
        otherwise Unreachable();
    return r;
```

aarch64/functions/sysregisters/PRBARnType

```
type PRBARnType;
```

aarch64/functions/sysregisters/PRLARn

```
// PRLARn[] - non-assignment form
// =============================

PRLARnType PRLARn[bits(2) regime, integer index]
    bits(64) r;
    case regime of
        when EL1  r = PRLARn_EL1[index];
        when EL2  r = PRLARn_EL2[index];
        otherwise Unreachable();
    return r;
```

aarch64/functions/sysregisters/PRLARnType

```
type PRLARnType;
```

aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        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;
        otherwise Unreachable();
```

```
        return r;

    // VBAR[] - non-assignment form
    // ==========================

    bits(64) VBAR[]
        return VBAR[S1TranslationRegime()];
```

aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================
// Checks if an AArch64 MSR, MRS or SYS instruction is allowed from
// the current exception level and security state. Also checks for
// traps by TIDCP to IMPLEMENTATION DEFINED registers and for NV access.

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                          bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    need_secure = FALSE;
    case op1 of
        when '00x'
            min_EL = EL1;
        when '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;

    if UInt(PSTATE.EL) < UInt(min_EL) then
        UNDEFINED;
    elsif need_secure && !IsSecure() then
        UNDEFINED;
```

aarch64/functions/system/AArch64.ExecutingATS1xPInstr

```
// AArch64.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1  = instr<16+:3>;
        CRn  = instr<12+:4>;
        CRm  = instr<8+:4>;
        op2  = instr<5+:3>;
        return op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'};
    else
        return FALSE;
```

aarch64/functions/system/AArch64.SysInstr

```
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

aarch64/functions/system/AArch64.SysInstrWithResult

```
// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);
```

aarch64/functions/system/AArch64.SysRegRead

```
// Read from a system register and return the contents of the register.
bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2);
```

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);
```

I1.1.4 aarch64/instrs

This section includes the following pseudocode functions:

aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// ========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)
```

```
                          SynchronizeContext();

                          sync_errors = HaveIESB() && SCTLR[].IESB == '1';
                          if sync_errors then
                              SynchronizeErrors();
                              iesb_req = TRUE;
                              TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
                          // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
                          SetPSTATEFromPSR(spsr);
                          ClearExclusiveLocal(ProcessorID());
                          SendEventLocal();

                          if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
                              // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
                              new_pc<63:32> = bits(32) UNKNOWN;
                              new_pc<1:0> = bits(2) UNKNOWN;
                          elsif UsingAArch32() then                 // Return to AArch32
                              // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set
state
                              if PSTATE.T == '1' then
                                  new_pc<0> = '0';                  // T32
                              else
                                  new_pc<1:0> = '00';               // A32
                          else                                      // Return to AArch64
                              // ELR_ELx[63:56] might include a tag
                              new_pc = AArch64.BranchAddr(new_pc);

                          if UsingAArch32() then
                              // 32 most significant bits are ignored.
                              BranchTo(new_pc<31:0>, BranchType_ERET);
                          else
                              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 exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;
```

```
                    case exttype of
                        when ExtendType_SXTB unsigned = FALSE; len = 8;
                        when ExtendType_SXTH unsigned = FALSE; len = 16;
                        when ExtendType_SXTW unsigned = FALSE; len = 32;
                        when ExtendType_SXTX unsigned = FALSE; len = 64;
                        when ExtendType_UXTB unsigned = TRUE;  len = 8;
                        when ExtendType_UXTH unsigned = TRUE;  len = 16;
                        when ExtendType_UXTW unsigned = TRUE;  len = 32;
                        when ExtendType_UXTX unsigned = TRUE;  len = 64;

                    // Note the extended width of the intermediate value and
                    // that sign extension occurs from bit <len+shift-1>, not
                    // from bit <len-1>. This is equivalent to the instruction
                    //    [SU]BFIZ Rtmp, Rreg, #shift, #len
                    // It may also be seen as a sign/zero extend followed by a shift:
                    //    LSL(Extend(val<len-1:0>, N, unsigned), shift);

                    len = Min(len, N - shift);
                    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

aarch64/instrs/extendreg/ExtendType

```
 enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                          ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
 enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                          FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
 enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                          FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

aarch64/instrs/float/convert/fpconvop/FPConvOp

```
 enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                          FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
                          , FPConvOp_CVT_FtoI_JS
 };
```

aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;
```

```
                       // must not match UXTx/SXTx alias
                       if immr == '000000' then
                           // must not match 32-bit UXT[BH] or SXT[BH]
                           if sf == '0' && imms IN {'000111', '001111'} then
                               return FALSE;
                           // must not match 64-bit SXT[BHW]
                           if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
                               return FALSE;

                       // must be UBFX/SBFX alias
                       return TRUE;
```

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 UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;     // 6-bit subtract with borrow

    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 != '1xxxxxx' then
    return FALSE;
if sf == '0' && immN:imms != '00xxxxx' then
    return FALSE;

// for MOVZ must contain no more than 16 ones
if S < 16 then
    // ones must not span halfword boundary when rotated
    return (-R MOD 16) <= (15 - S);

// for MOVN must contain no more than 16 zeros
if S >= width - 15 then
    // zeros must not span halfword boundary when rotated
    return (R MOD 16) <= (S - (width - 15));

return FALSE;
```

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 shiftype, integer amount)
    bits(N) result = X[reg];
    case shiftype 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        // Data Synchronization Barrier
                           ,  MemBarrierOp_DMB        // Data Memory Barrier
                           ,  MemBarrierOp_ISB        // Instruction Synchronization Barrier
                           ,  MemBarrierOp_SSBB       // Speculative Synchronization Barrier to VA
                           ,  MemBarrierOp_PSSBB      // Speculative Synchronization Barrier to PA
                           ,  MemBarrierOp_SB         // Speculation Barrier
                           };
```

aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_TSB,
    SystemHintOp_CSDB
};
```

aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         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 1010 001' return Sys_DC;   // CVAC
        when '000 0111 1010 010' return Sys_DC;   // CSW
        when '011 0111 1011 001' return Sys_DC;   // CVAU
        when '011 0111 1110 001' return Sys_DC;   // CIVAC
        when '000 0111 1110 010' return Sys_DC;   // CISW
        when '011 0111 1101 001' return Sys_DC;   // CVADP
        when '000 0111 0001 000' return Sys_IC;   // IALLUIS
        when '000 0111 0101 000' return Sys_IC;   // IALLU
        when '011 0111 0101 001' return Sys_IC;   // IVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
        when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
        when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE1
        when '000 1000 0111 101' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
```

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/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<esize-1:0>;

    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};
```

I1.1.5   aarch64/translation

This section includes the following pseudocode functions:

---

aarch64/translation/attrs/AArch64.CombineS1S2Desc

```
// AArch64.CombineS1S2Desc()
// ========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch64.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc, AccType
s2acctype)

    AddressDescriptor result;
    result.paddress = s2desc.paddress;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    else
        result.fault = AArch64.NoFault();
        fwb_enabled = apply_force_writeback && s2desc.memattrs.is_fwb;
        if s2desc.memattrs.memtype == MemType_Device || (!fwb_enabled && s1desc.memattrs.memtype ==
MemType_Device) then
            result.memattrs.memtype = MemType_Device;
            if s1desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s2desc.memattrs.device;
            elsif s2desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s1desc.memattrs.device;
            else                    // Both Device
                result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
```

```
                                                       s2desc.memattrs.device);
            else
                result.memattrs.memtype = MemType_Normal;
                result.memattrs.device = DeviceType UNKNOWN;
                if fwb_enabled then
                    // Check and apply cacheability condition
                    if S2CacheDisabled(s2acctype) then
                        result.memattrs.inner.attrs = MemAttr_NC;
                        result.memattrs.inner.hints = MemHint_No;
                    else
                        result.memattrs.inner.attrs = MemAttr_WB;    // force Write-back
                        if s1desc.memattrs.inner.attrs == MemAttr_NC
                            || s1desc.memattrs.outer.attrs == MemAttr_NC then
                            result.memattrs.inner.hints = MemHint_RWA;
                        else
                            result.memattrs.inner.hints = s1desc.memattrs.inner.hints;
                    result.memattrs.outer = result.memattrs.inner;
                else
                    result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner,
s2desc.memattrs.inner, s2acctype);
                    result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer,
s2desc.memattrs.outer, s2acctype);
                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;
```

aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                            bits(52) ipaddress, integer level,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, boolean UNKNOWN,  level, acctype, iswrite,
                                        secondstage, s2fs1walk);
    else
        addrdesc.memattrs.memtype = 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.memtype = 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.memtype = 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

        if ((HCR_EL2.VM == '1' || HCR_EL2.DC == '1') &&
            (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && HCR_EL2.TGE == '0'))) then
            return memattrs;
        else
            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, (acctype == AccType_IFETCH), PSTATE.EL);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(52) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,boolean UNKNOWN,  level, acctype,
                                                          iswrite, secondstage, s2fs1walk);

        return result;

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.memtype = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;       // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
```

```
        elsif acctype != AccType_IFETCH then
            // Treat data as Device
            result.addrdesc.memattrs.memtype = 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.memtype = 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.address = vaddress<51:0>;
        result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
        result.addrdesc.fault = AArch64.NoFault();

        result.descupdate.AF = FALSE;
        result.descupdate.AP = FALSE;
        result.descupdate.descaddr = result.addrdesc;

        return result;
```

aarch64/translation/attrs/AArch64.ValidateAddressS1Off

```
// AArch64.ValidateAddressS1Off()
// ==============================
// Called for stage 1 translations when translation is disabled to supply a
// default translation.

MPURecord AArch64.ValidateAddressS1Off(bits(64) inputaddr, AccType acctype,
                                       boolean iswrite)
    MPURecord result;

    if !IsZero(inputaddr<63:PAMax()>) then
        level = 0;
        address = bits(52) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(address,boolean UNKNOWN,
                                                         level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    default_cacheable  = HasS2Translation() && (HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheability settings
        result.addrdesc.memattrs.memtype             = MemType_Normal;
```

```
                      result.addrdesc.memattrs.inner.attrs   = MemAttr_WB;        // Write-back
                      result.addrdesc.memattrs.inner.hints   = MemHint_RWA;
                      result.addrdesc.memattrs.shareable     = FALSE;
                      result.addrdesc.memattrs.outershareable = FALSE;

              elsif acctype != AccType_IFETCH then
                  // Treat data as Device
                  result.addrdesc.memattrs.memtype = 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.memtype = 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.addrdesc.paddress.address = inputaddr<51:0>;
          result.addrdesc.paddress.NS = '0';
          result.addrdesc.fault = AArch64.NoFault();
          return result;
```

aarch64/translation/checks/AArch64.AccessIsPrivileged

```
// AArch64.AccessIsPrivileged()
// ============================

boolean AArch64.AccessIsPrivileged(AccType acctype)

    el = AArch64.AccessUsesEL(acctype);

    if el == EL0 then
        ispriv = FALSE;
    elsif el == EL3 then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

aarch64/translation/checks/AArch64.AccessUsesEL

```
// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

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 enable_wxn_pan,
                                    boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if (PSTATE.EL == EL0 ||
        PSTATE.EL == EL1) then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv = AArch64.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if !enable_wxn_pan then pan = '0';
        is_ldst  = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
        is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());
        if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn && enable_wxn_pan);
        priv_xn = perms.pxn == '1' || (priv_w && wxn && enable_wxn_pan) || (user_w && enable_wxn_pan);

        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 && enable_wxn_pan);

    // Restriction on Secure instruction fetch

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 then
        // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
        // other than DC IVAC, which requires write permission, and operations executed at EL0,
        // which require read permission.
        fail = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(52) UNKNOWN;
```

```
                    return AArch64.PermissionFault(ipaddress,boolean UNKNOWN,  level, acctype,
                                          !failedread, secondstage, s2fs1walk);
        else
            return AArch64.NoFault();
```

aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// ===========================
// Permission checking from AArch64 stage 2 validation

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(52) ipaddress,
                                      integer level, AccType acctype,
                                      boolean iswrite, boolean NS, boolean s2fs1walk)
    assert (IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2))) &&
HasS2Translation();
        if PSTATE.EL IN {EL0, EL1} then
            r = perms.ap<1> == '1';
            w = perms.ap<2:1> == '01';
        else
            // Access from EL2
            r = TRUE;
            w = perms.ap<2> == '0';

        if HaveExtendedExecuteNeverExt() then
            case perms.xn:perms.xxn of
                when '00'  xn = FALSE;
                when '01'  xn = PSTATE.EL == EL1;
                when '10'  xn = TRUE;
                when '11'  xn = PSTATE.EL == EL0;
        else
            xn = perms.xn == '1';
        // Stage 1 walk is checked as a read, regardless of the original type
        if acctype == AccType_IFETCH && !s2fs1walk then
            fail = xn;
            failedread = TRUE;

        elsif iswrite && !s2fs1walk then
            fail = !w;
            failedread = FALSE;

        elsif acctype == AccType_DC && PSTATE.EL != EL0 && !s2fs1walk then
            // DC maintenance instructions operating by VA, with the exception of DC IVAC, do
            // not generate Permission faults from stage 2 validation, other than when
            // performing a stage 1 translation table walk.
            fail = FALSE;

        else
            fail = !r;
            failedread = TRUE;

        if fail then
            secondstage = TRUE;
            return AArch64.PermissionFault(ipaddress,NS, 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, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.
```

```
FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress,  integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match 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",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch64.AccessIsPrivileged(acctype);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = vaddress;
        Halt(reason);
    elsif match then
```

```
            return AArch64.DebugFault(acctype, iswrite);
        else
            return AArch64.NoFault();
```

aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// ========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress,boolean NS, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// ========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress,boolean NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress, boolean UNKNOWN, level, acctype,
iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// ============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
```

```
        return AArch64.CreateFaultRecord(faulttype, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
extflag,
                                        errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress, boolean UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(bits(52) ipaddress,boolean NS, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(52) ipaddress, boolean NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
```

```
            errortype = bits(2) UNKNOWN;
        return AArch64.CreateFaultRecord(Fault_Translation, ipaddress, NS, level, acctype, iswrite,
                                    extflag, errortype, 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)

    boolean hw_update_AF = FALSE;
    boolean hw_update_AP = FALSE;

    // Check if access flag can be updated
    // Address translation instructions are permitted to update AF but not required
    if result.AF then
        if fault.statuscode == Fault_None || ConstrainUnpredictable() == Constraint_TRUE then
            hw_update_AF = TRUE;

    if result.AP && fault.statuscode == Fault_None then
        write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW,AccType_ORDEREDRW,
AccType_ORDEREDATOMICRW }) && !s2fs1walk;
        hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC, AccType_DC_UNPRIV})) ||
hwupdatewalk;

    if hw_update_AF || hw_update_AP then
        if secondstage || !HasS2Translation() then
            descaddr2 = result.descaddr;
        else
            assert hwupdatewalk == FALSE;
            descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8,
hwupdatewalk);
            if IsFault(descaddr2) then
                return descaddr2.fault;

        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        desc = _Mem[descaddr2, 8, accdesc];
        el = AArch64.AccessUsesEL(acctype);
        case el of
            when EL2
                reversedescriptors = SCTLR_EL2.EE == '1';
            otherwise
                reversedescriptors = SCTLR_EL1.EE == '1';
        if reversedescriptors then
            desc = BigEndianReverse(desc);

        if hw_update_AF then
            desc<10> = '1';
        if hw_update_AP then
            desc<7> = (if secondstage then '1' else '0');

        _Mem[descaddr2,8,accdesc] = if reversedescriptors then BigEndianReverse(desc) else 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';

    TLBRecord S1;
    S1.addrdesc.fault = AArch64.NoFault();
    ipaddress = bits(52) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                              // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, TRUE, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if (((!wasaligned && acctype != AccType_IFETCH) || acctype == AccType_DCZVA)
        && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device) then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype,
                                                    TRUE,
                                                    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.memtype == 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.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.SecondStageValidate(S1, vaddress, acctype, iswrite, wasaligned, size, s2fs1walk);
```

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.FullValidate(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/validation/AArch64.ComputeS2NSbit

```
// AArch64.ComputeS2NSbit()
// ========================
// Compute stage 2 NS bit for EL1&0 translation regime.

bit AArch64.ComputeS2NSbit(bit s1_ns, bit s2_ns)
    nsaccess = s2_ns OR s1_ns;
    if nsaccess == '0' then
        nsaccess = VSTCR_EL2.SA;
    else
        if VSTCR_EL2.SA == '1' then nsaccess = '1';
        else nsaccess = VTCR_EL2.NSA;
    return nsaccess;
```

aarch64/translation/validation/AArch64.FirstStageValidate

```
// AArch64.FirstStageValidate()
// ============================
// Performs stage 1 validation for the memory access using MPU.

AddressDescriptor AArch64.FirstStageValidate(bits(64) vaddress, AccType acctype,
                        boolean iswrite, boolean wasaligned, integer size)
    // Check whether stage 1 is enabled
    if HasS2Translation() then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR[].M == '1';
        br_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR[].BR == '1';
    else
        s1_enabled = SCTLR[].M == '1';
        br_enabled = SCTLR[].BR == '1';

    secondstage  = FALSE;
    s1_nonsecure = FALSE;
    s2fs1walk    = FALSE;
    level        = 0;

    el = AArch64.AccessUsesEL(acctype);
    top = AddrTop(vaddress, (acctype == AccType_IFETCH), el);
    inputaddr = ZeroExtend(vaddress<top:0>,64);

    if s1_enabled then
        S1 = AArch64.MPUValidate(inputaddr, acctype, s1_nonsecure, iswrite, secondstage,
                            size, br_enabled, s2fs1walk);
        permissioncheck = TRUE;
        enable_wxn_pan  = TRUE;

    elsif br_enabled then
```

```
                S1 = AArch64.GetDefaultMemoryAttr(inputaddr, acctype, s1_nonsecure, iswrite,
                              secondstage,s2fs1walk);
                permissioncheck = TRUE;
                enable_wxn_pan = FALSE;
            else
                S1 = AArch64.ValidateAddressS1Off(inputaddr, acctype, iswrite);
                permissioncheck = FALSE;

            // Check for unaligned data accesses to Device memory
            if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
                && S1.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S1.addrdesc) then
                S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

            if !IsFault(S1.addrdesc) && permissioncheck then
                S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, level,
                                                            S1.addrdesc.paddress.NS, acctype,
                                                            enable_wxn_pan,
                                                            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.memtype == MemType_Device &&
                acctype == AccType_IFETCH) then
                S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress,
                                    S1.addrdesc.paddress.address, level, acctype, iswrite,
                                    secondstage, s2fs1walk);
            return S1.addrdesc;
```

aarch64/translation/validation/AArch64.FullValidate

```
// AArch64.FullValidate()
// =====================
// Perform both stage 1 and stage 2 address validation for the current memory access
// regime.

AddressDescriptor AArch64.FullValidate(bits(64) vaddress, AccType acctype,
                                       boolean iswrite, boolean wasaligned, integer size)
    // First stage validation
    if IsStage1VMSA() then
        S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    else
        S1 = AArch64.FirstStageValidate(vaddress, acctype, iswrite, wasaligned, size);

    // Second Stage Validation
    if !IsFault(S1) && HasS2Translation() then
        s2fs1walk = FALSE;
        result = AArch64.SecondStageValidate(S1, vaddress, acctype, iswrite, wasaligned,
                                             size, s2fs1walk);
    else
        result = S1;

    return result;
```

aarch64/translation/validation/AArch64.GetDefaultMemoryAttr

```
// AArch64.GetDefaultMemoryAttr()
// =============================
// Get default memory attributes.

MPURecord AArch64.GetDefaultMemoryAttr(bits(64) inputaddr, AccType acctype,
                                       boolean s1_nonsecure, boolean iswrite,
                                       boolean secondstage, boolean s2fs1walk)

    MPURecord memrecord;

    level = 0;
```

```
                address = inputaddr<51:0>;
                s1_ns = if s1_nonsecure then '1' else '0';
                NS = s1_ns == '1';

                if boolean IMPLEMENTATION_DEFINED "Default Memory Map" then
                    // Check whether address is greater than implemented PA range
                    if !IsZero(inputaddr<63:PAMax()>) then
                        memrecord.addrdesc.fault =  AArch64.TranslationFault(address<51:0>,
                                                                      NS, level,
                                                                      acctype, iswrite,
                                                                      secondstage, s2fs1walk);
                        memrecord.perms = Permissions UNKNOWN;
                        return memrecord;

                    (is_valid, ns, memattrs, ap, xn, fwb_encoding, xxn) = __BackgroundMemoryAttr(inputaddr, acctype,
                                                                                secondstage);
                    if !is_valid || (secondstage && VSTCR_EL2.SC == '1' && s1_ns != ns) then
                        memrecord.addrdesc.fault = AArch64.TranslationFault(address, NS,
                                                                      level, acctype, iswrite,
                                                                      secondstage, s2fs1walk);
                        memrecord.perms = Permissions UNKNOWN;
                    else

                        // Check whether FWB is enabled
                        if secondstage then
                            memattrs.is_fwb = fwb_encoding && HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
                            if memattrs.is_fwb then assert ((memattrs.inner.attrs == MemAttr_WB) &&
(memattrs.outer.attrs == MemAttr_WB));
                        // If FWB is enabled, then cacheability control is applied after applying FWB in
                        // CombineS1S2Desc(). Else cacheability is applied in here
                        if memattrs.memtype == MemType_Normal
                            && ((!secondstage && S1CacheDisabled(acctype)) || (secondstage &&
S2CacheDisabled(acctype) && !memattrs.is_fwb)) then
                            memattrs.inner.attrs = MemAttr_NC;
                            memattrs.inner.hints = MemHint_No;
                            memattrs.outer = memattrs.inner;

                        memrecord.addrdesc.memattrs = memattrs;
                        memrecord.perms.ap          = ap;
                        memrecord.perms.xn          = xn;
                        memrecord.perms.pxn         = memrecord.perms.xn;
                        if !secondstage then
                            memrecord.addrdesc.paddress.NS = ns;
                        else
                            memrecord.addrdesc.paddress.NS = AArch64.ComputeS2NSbit(s1_ns, ns);
                            if HaveExtendedExecuteNeverExt() then memrecord.perms.xxn = xxn;
                        memrecord.addrdesc.paddress.address = address;
                        memrecord.addrdesc.fault = AArch64.NoFault();
                else
                    Unreachable();

                return memrecord;
```

aarch64/translation/validation/AArch64.MPUValidate

```
// AArch64.MPUValidate()
// =====================
// Returns the result of MPU checks on the address.

MPURecord AArch64.MPUValidate(bits(64) inputaddr, AccType acctype,
                              boolean s1_nonsecure, boolean iswrite, boolean secondstage,
                              integer size, boolean br_enabled, boolean s2fs1walk)

    MPURecord memrecord;
    address = inputaddr<51:0>;

    if !secondstage then assert !s1_nonsecure;
```

```
                    // Select the MPU for memory validation
                    if S1TranslationRegime() == EL1 && !secondstage then
                        mpu_el = EL1;
                    else
                        mpu_el = EL2;

                    level = 0;
                    matched = FALSE;
                    num_regions = UInt(MPUIR[mpu_el].REGION);
                    s1_ns = if s1_nonsecure then '1' else '0';

                    // Check whether address is greater than implemented PA range
                    if !IsZero(inputaddr<63:PAMax()>) then
                        memrecord.addrdesc.fault = AArch64.TranslationFault(address<51:0>,
                                                                    s1_nonsecure, level,
                                                                    acctype, iswrite,
                                                                    secondstage, s2fs1walk);
                        memrecord.perms = Permissions UNKNOWN;
                        return memrecord;

                    if num_regions != 0 then
                        for r = 0 to num_regions-1
                            prbar = PRBARn[mpu_el, r];
                            prlar = PRLARn[mpu_el, r];

                            if prlar.EN == '1' then
                                base  = ZeroExtend(prbar.BASE  : Zeros(6), 64);
                                limit = ZeroExtend(prlar.LIMIT : Ones(6), 64);

                                // Check for a matching MPU region
                                if UInt(inputaddr) >= UInt(base) && UInt(inputaddr) <= UInt(limit) then
                                    // Check for multiple region match
                                    if matched then
                                        memrecord.addrdesc.fault = AArch64.TranslationFault(address,
                                                                                    s1_nonsecure,
                                                                                    level, acctype,
                                                                                    iswrite,
                                                                                    secondstage,
                                                                                    s2fs1walk);
                                        memrecord.perms = Permissions UNKNOWN;
                                        return memrecord;
                                    else
                                        matched = TRUE;
                                        ns      =  prlar.NS;

                                        if secondstage && VSTCR_EL2.SC == '1' && s1_ns != ns then
                                            memrecord.addrdesc.fault = AArch64.TranslationFault(address,
                                                                                        s1_nonsecure,
                                                                                        level,
                                                                                        acctype,
                                                                                        iswrite,
                                                                                        secondstage,
                                                                                        s2fs1walk);
                                            memrecord.perms = Permissions UNKNOWN;
                                            return memrecord;

                                        if !secondstage then
                                            memrecord.addrdesc.paddress.NS = ns;
                                        else
                                            memrecord.addrdesc.paddress.NS = AArch64.ComputeS2NSbit(s1_ns, ns);
                                        memrecord.addrdesc.paddress.address = address;
                                        memrecord.perms.ap<2:1> = prbar.AP;
                                        memrecord.perms.ap<0> = '1';
                                        memrecord.perms.xn = prbar.XN<1>;
                                        if secondstage && HaveExtendedExecuteNeverExt() then
                                            memrecord.perms.xxn = prbar.XN<0>;
                                        memrecord.perms.pxn = memrecord.perms.xn;
                                        attrIndx = prlar.AttrIndx;
```

```
                                    sh = prbar.SH;
                                    if !secondstage then
                                        memrecord.addrdesc.memattrs = AArch64.S1AttrDecode(sh,
                                                                                            attrIndx,
                                                                                            acctype);
                                    else
                                        memrecord.addrdesc.memattrs = S2MPUAttrDecode(sh, attrIndx,
                                                                                      acctype);

                if matched then
                    memrecord.addrdesc.fault = AArch64.NoFault();
                elsif br_enabled && HighestELAccess(secondstage) && AArch64.AccessIsPrivileged(acctype) then
                    memrecord = AArch64.GetDefaultMemoryAttr(inputaddr, acctype, s1_nonsecure,
                                                             iswrite, secondstage, s2fs1walk);
                    if IsFault(memrecord.addrdesc) then return memrecord;
                else
                    // No MPU match or background region enabled
                    memrecord.addrdesc.fault = AArch64.TranslationFault(address, s1_nonsecure, level,
                                                                        acctype, iswrite, secondstage,
                                                                        s2fs1walk);
                    memrecord.perms = Permissions UNKNOWN;
                    return memrecord;

                if secondstage && HaveCommonNotPrivateTransExt() && IsStage1VMSA() then
                    memrecord.CnP = VSCTLR_EL2.CnP;
                return memrecord;
```

aarch64/translation/validation/AArch64.SecondStageValidate

```
// AArch64.SecondStageValidate()
// =============================
// Perform a stage 2 validation for memory access.

AddressDescriptor AArch64.SecondStageValidate(AddressDescriptor S1, bits(64) vaddress,
                                              AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size,
                                              boolean s2fs1walk)
        assert HasS2Translation();

        s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
        br_enabled = SCTLR_EL2.BR == '1';
        mpu_enabled = SCTLR_EL2.M == '1';
        secondstage = TRUE;
        level = 0;

        if s2_enabled then                              // Second stage enabled
            s1_nonsecure = S1.paddress.NS == '1';
            inputaddr = ZeroExtend(S1.paddress.address, 64);

            if mpu_enabled then
                S2 = AArch64.MPUValidate(inputaddr, acctype, s1_nonsecure,  iswrite,
                                         secondstage, size, br_enabled, s2fs1walk);
                permissioncheck = TRUE;
            elsif br_enabled then
                S2 = AArch64.GetDefaultMemoryAttr(inputaddr, acctype, s1_nonsecure, iswrite,
                                                  secondstage, s2fs1walk);
                permissioncheck = TRUE;
            else
                // If HCR_EL2.VM = 1 and SCTLR_EL2.{M, BR} = {0,0}, then the behavior is CONSTRAINED
UNPREDICTABLE.
                c = ConstrainUnpredictable();
                assert c IN {Constraint_MPU_FAULT, Constraint_MPU_ATTR_UNKNOWN};

                if c == Constraint_MPU_FAULT then
                    S2.addrdesc.fault = AArch64.TranslationFault(inputaddr<51:0>,
                                                                 s1_nonsecure, level,
                                                                 acctype, iswrite,
```

```
                                                        secondstage, s2fs1walk);
                S2.perms = Permissions UNKNOWN;

            else
                S2.addrdesc.memattrs = MemoryAttributes UNKNOWN;
                S2.perms = Permissions UNKNOWN;

            // Note: If the output address from the stage 1 translation is larger than the implemented
            // PA size, then a stage 1 Address size fault is generated for the level of the
            // stage 1 translation that generated the output address. This can happen if EL1&0
            // stage 1 translation is running VMSA. In this case Address size fault is generated
            // in TranslationTableWalk().

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
                && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        // Check for permissions on Stage2 validation
        if !IsFault(S2.addrdesc) && permissioncheck then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, S1.paddress.address,
                                                          level, acctype, iswrite,
                                                          s1_nonsecure, 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.memtype == MemType_Device
                && acctype == AccType_IFETCH) then

            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress,
                                                    S1.paddress.address, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
                S2.addrdesc.memattrs.memtype == MemType_Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(S1.paddress.address, s1_nonsecure, level,
acctype,
                                                            iswrite, secondstage, s2fs1walk);

        result = AArch64.CombineS1S2Desc(S1, S2.addrdesc, acctype);
    else
        // If stage 2 translation is disabled and the output address from the stage 1
        // translation is larger than the implemented PA size, then a stage 1
        // Address  size fault is generated for the level of the stage 1 translation
        // that generated the output address.
        // This can happen if EL1&0 stage 1 translation is running VMSA. In this case
        // Address size fault is generated in TranslationTableWalk().
        result = S1;

    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(52) ipaddress, boolean s1_nonsecure, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
```

```
            if !secondstage then
                assert !ELUsingAArch32(S1TranslationRegime());
            else
                assert (IsSecureEL2Enabled() || (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
            bit nswalk;                           // Stage 2 translation table walks are to Secure or to Non-secure PA
space

            result.descupdate.AF = FALSE;
            result.descupdate.AP = FALSE;

            descaddr.memattrs.memtype = MemType_Normal;

            // Derived parameters for the page table walk:
            //  grainsize = Log2(Size of Table)        - Size of Table is 4KB, 16KB or 64KB in AArch64
            //  stride = Log2(Address per Level)        - Bits of address consumed at each level
            //  firstblocklevel = First level where a block entry is allowed
            //  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
            //  inputsize = Log2(Size of Input Address) - Input Address size in bits
            //  level = Level to start walk from
            // This means that the number of levels after start level = 3-level

            if !secondstage then
                // First stage translation
                inputaddr = ZeroExtend(vaddress);
                el = AArch64.AccessUsesEL(acctype);
                isprivileged = AArch64.AccessIsPrivileged(acctype);
                top = AddrTop(inputaddr, (acctype == AccType_IFETCH), el);
                if el IN {EL0, EL1} then
                    if inputaddr<top> == '0' then
                        inputsize = 64 - UInt(TCR_EL1.T0SZ);
                        largegrain = TCR_EL1.TG0 == '01';
                        midgrain = TCR_EL1.TG0 == '10';
                        inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;

                        if !Have52BitVAExt() && inputsize > inputsize_max then
                            c = ConstrainUnpredictable();
                            assert c IN {Constraint_FORCE, Constraint_FAULT};
                            if c == Constraint_FORCE then inputsize = inputsize_max;

                        inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
48);
                        if inputsize < inputsize_min then
                            c = ConstrainUnpredictable();
                            assert c IN {Constraint_FORCE, Constraint_FAULT};
                            if c == Constraint_FORCE then inputsize = inputsize_min;
                        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
IsZero(inputaddr<top:inputsize>);
                        disabled = TCR_EL1.EPD0 == '1' || (!isprivileged && HaveE0PDExt() && TCR_EL1.E0PD0 ==
'1');
                        baseregister = TTBR0_EL1;
                        descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0,
secondstage);
                        hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD0 == '1';
                    else
                        inputsize = 64 - UInt(TCR_EL1.T1SZ);
                        largegrain = TCR_EL1.TG1 == '11';        // TG1 and TG0 encodings differ
                        midgrain = TCR_EL1.TG1 == '01';
                        inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;

                        if !Have52BitVAExt() && inputsize > inputsize_max then
                            c = ConstrainUnpredictable();
                            assert c IN {Constraint_FORCE, Constraint_FAULT};
```

```
                                if c == Constraint_FORCE then inputsize = inputsize_max;

                            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
             48);
                            if inputsize < inputsize_min then
                                c = ConstrainUnpredictable();
                                assert c IN {Constraint_FORCE, Constraint_FAULT};
                                if c == Constraint_FORCE then inputsize = inputsize_min;
                            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
             IsOnes(inputaddr<top:inputsize>);
                            disabled = TCR_EL1.EPD1 == '1' || (!isprivileged && HaveE0PDExt() && TCR_EL1.E0PD1 ==
             '1');
                            baseregister = TTBR1_EL1;
                            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1,
             secondstage);
                            hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD1 == '1';
                        ps = TCR_EL1.IPS;
                        reversedescriptors = SCTLR_EL1.EE == '1';
                        lookupsecure = IsSecure();
                        singlepriv = FALSE;
                        update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
                        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
                    else
                        Unreachable();
                    if largegrain then
                        grainsize = 16;                                      // Log2(64KB page size)
                        firstblocklevel = (if Have52BitPAExt() then 1 else 2);    // Largest block is 4TB (2^42
             bytes) for 52 bit PA

                                                                              // and 512MB (2^29 bytes)
             otherwise
                    elsif 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 - (1 + ((inputsize - grainsize - 1) DIV stride));

                else
                    Unreachable();
                if !basefound || disabled then
                    level = 0;            // AArch32 reports this as a level 1 fault
                    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure,  level, acctype,
             iswrite,
                                                                     secondstage, s2fs1walk);
                    return result;

                case ps of
                    when '000'  outputsize = 32;
                    when '001'  outputsize = 36;
                    when '010'  outputsize = 40;
                    when '011'  outputsize = 42;
                    when '100'  outputsize = 44;
                    when '101'  outputsize = 48;
                    when '110'  outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
                    otherwise   outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address
             size value";

                if outputsize > PAMax() then outputsize = PAMax();

                if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
                    level = 0;
                    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure,  level, acctype,
             iswrite,
```

```
                                                        secondstage, s2fs1walk);
            return result;

        // Bottom bound of the Base address is:
        //     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
        // Number of entries in starting level table =
        //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
        baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);   // Log2(Num of entries*8)
        if outputsize == 52 then
            z = (if baselowerbound < 6 then 6 else baselowerbound);
            baseaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
        else
            baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

        ns_table = if lookupsecure then '0' else '1';
        ap_table = '00';
        xn_table = '0';
        pxn_table = '0';

        addrselecttop = inputsize - 1;

        repeat
            addrselectbottom = (3-level)*stride + grainsize;

            bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
            descaddr.paddress.address = baseaddress OR index;
            descaddr.paddress.NS = if secondstage then nswalk else ns_table;

            // If there are two stages of translation, then the first stage table walk addresses
            // are themselves subject to translation
            if secondstage || !HasS2Translation() then
                descaddr2 = descaddr;
            else
                hwupdatewalk = FALSE;
                descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
                // Check for a fault on the stage 2 walk
                if IsFault(descaddr2) then
                    result.addrdesc.fault = descaddr2.fault;
                    return result;

            // Update virtual address for abort functions
            descaddr2.vaddress = ZeroExtend(vaddress);

            accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
            desc = _Mem[descaddr2, 8, accdesc];

            if reversedescriptors then desc = BigEndianReverse(desc);

            if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 ||
                                                 (HaveBlockBBM() && IsBlockDescriptorNTBitValid() &&
desc<16> == '1'))) then
                 // Fault (00), Reserved (10), Block (01) at level 3, or Block(01) with nT bit set.
                result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
                return result;

            // Valid Block, Page, or Table entry
            if desc<1:0> == '01' || level == 3 then                // Block (01) or Page (11)
                blocktranslate = TRUE;
            else                                                   // Table (11)
                if (outputsize < 52 && largegrain && (PAMax() == 52 ||
                    boolean IMPLEMENTATION_DEFINED "Address Size Fault on LPA descriptor bits [15:12]") &&
                    !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(desc<47:outputsize>)) then
                    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure,  level,
acctype,
                                                        iswrite, secondstage, s2fs1walk);
                    return result;
```

```
                    if outputsize == 52 then
                        baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
                    else
                        baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));
                    if !secondstage then
                        // Unpack the upper and lower table attributes
                        ns_table    = ns_table    OR desc<63>;
                    if !secondstage && !hierattrsdisabled then
                        ap_table<1> = ap_table<1> OR desc<62>;        // read-only

                        xn_table    = xn_table    OR desc<60>;
                        // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
                        if !singlepriv then
                            pxn_table    = pxn_table    OR desc<59>;
                            ap_table<0> = ap_table<0> OR desc<61>;    // privileged

                level = level + 1;
                addrselecttop = addrselectbottom - 1;
                blocktranslate = FALSE;
        until blocktranslate;

        // Check block size is supported at this level
        if level < firstblocklevel then
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;

        // Check for misprogramming of the contiguous bit
        if largegrain then
            num_ch_entries = 5;
        elsif midgrain then
            num_ch_entries = if level == 3 then 7 else 5;
        else
            num_ch_entries = 4;

        contiguousbitcheck = inputsize < (addrselectbottom + num_ch_entries);

        if contiguousbitcheck && desc<52> == '1' then
            if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
                result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                                iswrite, secondstage, s2fs1walk);
            return result;

        // Unpack the descriptor into address and upper and lower block attributes
        if largegrain then
            outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
        else
            outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);

        // When 52-bit PA is supported, for 64 Kbyte translation granule,
        // block size might be larger than the supported output address size
        if ((outputsize < 52 && !IsZero(outputaddress<51:48>) && largegrain && (PAMax() == 52 ||
            boolean IMPLEMENTATION_DEFINED "Address Size Fault on LPA descriptor bits [15:12]")) ||
            (outputsize < 48 && !IsZero(outputaddress<47:outputsize>))) then
            result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure,  level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;

        // Check Access Flag
        if desc<10> == '0' then
            if !update_AF then
                result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress,s1_nonsecure,  level, acctype,
                                                               iswrite, secondstage, s2fs1walk);
                return result;
            else
                result.descupdate.AF = TRUE;

        if update_AP && desc<51> == '1' then
```

```
                    // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
                    if !secondstage && desc<7> == '1' then
                        desc<7> = '0';
                        result.descupdate.AP = TRUE;
                    elsif secondstage && desc<7> == '0' then
                        desc<7> = '1';
                        result.descupdate.AP = TRUE;

            // Required descriptor if AF or AP[2]/S2AP[2] needs update
            result.descupdate.descaddr = descaddr;

            xn = desc<54>;                                     // Bit[54] of the block/page descriptor
holds UXN
            pxn = desc<53>;                                    // Bit[53] of the block/page descriptor
holds PXN
            ap = desc<7:6>:'1';                                // Bits[7:6] of the block/page descriptor
hold AP[2:1]
            contiguousbit = desc<52>;
            nG = desc<11>;
            sh = desc<9:8>;
            memattr = desc<5:2>;                              // AttrIndx and NS bit in stage 1

            result.domain = bits(4) UNKNOWN;                  // Domains not used
            result.level = level;
            result.blocksize = 2^((3-level)*stride + grainsize);

            // Stage 1 translation regimes also inherit attributes from the tables
            if !secondstage then
                result.perms.xn     = xn OR xn_table;
                result.perms.ap<2>  = ap<2> OR ap_table<1>;          // Force read-only
                // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
                if !singlepriv then
                    result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
                    result.perms.pxn   = pxn OR pxn_table;
                    // Pages from Non-secure tables are marked non-global in Secure EL1&0
                    if IsSecure() then
                        result.nG = nG OR ns_table;
                    else
                        result.nG = nG;
                else
                    result.perms.ap<1> = '1';
                    result.perms.pxn   = '0';
                    result.nG          = '0';
                result.perms.ap<0>   = '1';
                result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
                result.addrdesc.paddress.NS = memattr<3> OR ns_table;
            else
                Unreachable();

        result.addrdesc.paddress.address = outputaddress;
        result.addrdesc.fault = AArch64.NoFault();
        result.contiguous = contiguousbit == '1';
        if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

        return result;
```

I1.2 Shared pseudocode

This section lists the pseudocodes that are common to execution in AArch64 state and in the Armv8-R AArch64 state. Armv8-R AArch64 supports AArch64 state and this document correlates with that on the Armv8-A profile. This document follows the structure of the *Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile*.

The functions listed in this section are identified only by a `FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

I1.2.1 shared/debug

This section includes the following pseudocode functions:

shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';          // Clear TX underrun flag
    EDSCR.RXO = '0';          // Clear RX overrun flag

    if Halted() then          // in Debug state
        EDSCR.ITO = '0';      // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool() then
        return;
    EDSCR.ERR = '0';          // Clear cumulative error flag

    return;
```

shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level
```

```
bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(boolean secure)
    if !secure || HaveSecureEL2Ext() then
        route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        Unreachable();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        return AllowExternalDebugAccess(IsAccessSecure());
    else
        return AllowExternalDebugAccess(ExternalSecureInvasiveDebugEnabled());

// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(boolean allow_secure)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalInvasiveDebugEnabled() then
        if allow_secure then
            return TRUE;
        else
```

```
                            return !IsSecure();
                else
                    return FALSE;
```

shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =======================
// Returns TRUE if an external debug interface access to the PMU registers is allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        return AllowExternalPMUAccess(IsAccessSecure());
    else
        return AllowExternalPMUAccess(ExternalSecureNoninvasiveDebugEnabled());

// AllowExternalPMUAccess()
// =======================
// Returns TRUE if an external debug interface access to the PMU registers is allowed for the given
// Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(boolean allow_secure)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalNoninvasiveDebugEnabled() then
        if allow_secure then
            return TRUE;
        else
            return !IsSecure();
    else
        return FALSE;
```

shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// =============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;
```

shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// ================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return (ExternalNoninvasiveDebugEnabled() && ExternalSecureNoninvasiveDebugEnabled());
```

shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// ================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented, otherwise this
// function is IMPLEMENTATION DEFINED.
```

```
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
// OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ==================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();
```

shared/debug/authentication/IsAccessSecure

```
// Returns TRUE when an access is Secure
boolean IsAccessSecure();
```

shared/debug/authentication/IsCorePowered

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();
```

shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// =====================
// Checks for an invalid state match that will generate Constrained Unpredictable behaviour, otherwise
// returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean
isbreakpnt)
    boolean reserved = FALSE;

    // Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;

    // EL3 is not implemented
```

```
        if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
            reserved = TRUE;

        // EL3 using AArch64 only
        if (!HaveEL(EL3) || HighestELUsingAArch32()) && HMC:SSC:PxC == '11000' then
            reserved = TRUE;

        // EL2 is not implemented
        if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
            reserved = TRUE;

        // Secure EL2 is not implemented
        if !HaveSecureEL2Ext() && (HMC:SSC:PxC) IN {'01100','10100','x11x1'} then
            reserved = TRUE;

        // Values that are not allocated in any architecture version
        if (HMC:SSC:PxC) IN {'01110','100x0','10110','11x10'} then
            reserved = TRUE;

        if reserved then
            // If parameters are set to a reserved type, behaves as either disabled or a defined type
            (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
            // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

        return (Constraint_NONE, SSC, HMC, PxC);
```

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 "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';          // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                            // See comments in EDITR[] (external write)
        ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
        X[1] = bits(64) 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 "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;
```

```
            if EDSCR.ERR == '1' then return value;          // Error flag set: no side-effects

            // The Software lock is OPTIONAL.
            if memory_mapped && EDLSR.SLK == '1' then        // Software lock locked: no side-effects
                return value;

            if underrun then
                EDSCR.TXU = '1';   EDSCR.ERR = '1';          // Underrun condition: block side-effects
                return value;                                // Return UNKNOWN

            EDSCR.TXfull = '0';
            if Halted() && EDSCR.MA == '1' then
                EDSCR.ITE = '0';                             // See comments in EDITR[] (external write)

                if !UsingAArch32() then
                    ExecuteA64(0xB8404401<31:0>);            // A64 "LDR W1,[X0],#4"
                else
                    ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
                // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
                if EDSCR.ERR == '1' then
                    EDSCR.TXfull = bit UNKNOWN;
                    DBGDTRTX_EL0 = bits(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';
                X[1] = bits(64) 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 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                  // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                       // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';          // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();
```

```
                    case target_el of
                        when EL1
                            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
                            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
                            else handle_el = EL1;

                        when EL2
                            if !HaveEL(EL2) then UNDEFINED;
                            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
                            elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
                            else handle_el = EL2;
                        when EL3
                            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
                            handle_el = EL3;
                        otherwise
                            Unreachable();

                    from_secure = IsSecure();
                    PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
                    if (HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0'))) then
                        PSTATE.PAN = '1';
                    ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
                    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;
                    if HaveUAOExt() then PSTATE.UAO = '0';

                    UpdateEDSCRFields();                          // Update EDSCR PE state flags
                    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
                    // SCTLR[].IESB might be ignored in Debug state.
                    if !ConstrainUnpredictableBool() then
                        sync_errors = FALSE;
                    if sync_errors then
                        SynchronizeErrors();
                    return;
```

shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool() then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';         // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;
```

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

            return;
```

shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

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 PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                     // Signal restarting
    EDESR<2:0> = '000';                          // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(32) spsr;

    new_pc = DLR_EL0;
    spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                      // Can update privileged bits, even at EL0

    if UsingAArch32() then
        if ConstrainUnpredictableBool() then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT);    // 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 PE state
        DisableITRAndResumeInstructionPrefetch();

        return;
```

shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    spsr = GetPSRFromPSTATE();

    if UsingAArch32() then
        // If entering from AArch32 state, spsr<21> is the DIT bit which has to be moved for DSPSR
        spsr<24> = spsr<21>;
        spsr<21> = PSTATE.SS;                 // Always save the SS bit

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                      // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';              // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //    are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                     // Signal entered Debug state
    UpdateEDSCRFields();                       // Update EDSCR PE 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;
    elsif 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 PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = '0';
        EDSCR.RW = '1111';
    return;
```

shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
```

```
// Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
// for the exception target.
base = if IsSecure() then 0 else 4;
if HaltingAllowed() then
    if HaveExtendedECDebugEvents() then
        exception_exit = !exception_entry;
        ctrl = EDECCR<UInt(PSTATE.EL) + base + 8>:EDECCR<UInt(PSTATE.EL) + base>;
        case ctrl of
            when '00'  halt = FALSE;
            when '01'  halt = TRUE;
            when '10'  halt = (exception_exit == TRUE);
            when '11'  halt = (exception_entry == TRUE);
    else
        halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
    if halt then Halt(DebugHalt_ExceptionCatch);
```

shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()

    if (HaveDoPD() && CTIDEVCTL.OSUCE == '1')
    then
        if !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 (HaveDoPD() && CTIDEVCTL.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 = OSLSR_EL1.OSLK;
    if HaltingAllowed() && EDSCR.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" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();            // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();
```

```
        if active && reset then                    // Coming out of reset with EDECR.SS set
            EDESR.SS = '1';
        elsif 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/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    case target of
        when EL3
            int_dis = EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled();
        when EL2
            int_dis = EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled();
        when EL1
            if IsSecure() then
                int_dis = EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled();
            else
                int_dis = EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled();
    return int_dis;
```

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 = CONTEXTIDR_EL1;
    pc_sample.has_el2 = EL2Enabled();

    if EL2Enabled() then
        if !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VSCTLR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VSCTLR_EL2.VMID;
```

```
                               pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
                               pc_sample.el0h = FALSE;
                    return;
```

shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
      boolean valid,
      bits(64) pc,
      bits(2) el,
      bit rw,
      bit ns,
      boolean has_el2,
      bits(32) contextidr,
      bits(32) contextidr_el2,
      boolean el0h,
      bits(16) vmid
)

PCSample pc_sample;
```

shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// ==============

bits(32) PMPCSR[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                 // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';       // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            PMPCSR.EL = pc_sample.el;
            PMPCSR.NS = pc_sample.ns;

            PMCID1SR = pc_sample.contextidr;
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
                            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32>  = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;

            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;

            PMVIDSR.VMID = bits(16) UNKNOWN;

    return sample;
```

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;
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;
        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) && MDSCR_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.
// Might return TRUE or FALSE if the previously executed instruction was an ISB or ERET executed
// in the active-not-pending state, or if another exception was taken before the Software Step
```

```
exception.
 // Returns FALSE otherwise, indicating that the previously executed instruction was executed in the
 // active-not-pending state, that is, the instruction was stepped.
 boolean SoftwareStep_DidNotStep();
```

shared/debug/softwarestep/SoftwareStep_SteppedEX

```
 // Returns a value that describes the previously executed instruction. The result is valid only if
 // SoftwareStep_DidNotStep() returns FALSE.
 // Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX that failed its condition
code test.
 // Otherwise returns TRUE if the instruction was a Load-Exclusive class instruction, and FALSE if the
 // instruction was not a Load-Exclusive class instruction.
 boolean SoftwareStep_SteppedEX();
```

I1.2.2   shared/exceptions

This section includes the following pseudocode functions:

- *shared/exceptions/exceptions/ConditionSyndrome.*
- *shared/exceptions/exceptions/Exception* on page I1-393.
- *shared/exceptions/exceptions/ExceptionRecord* on page I1-393.
- *shared/exceptions/exceptions/ExceptionSyndrome* on page I1-393.
- *shared/exceptions/traps/ReservedValue* on page I1-394.
- *shared/exceptions/traps/UnallocatedEncoding* on page I1-394.

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_CP15RTTrap,         // Trapped AArch32 MCR or MRC access to CP15
                       Exception_CP15RRTTrap,        // Trapped AArch32 MCRR or MRRC access to CP15
                       Exception_CP14RTTrap,         // Trapped AArch32 MCR or MRC access to CP14
                       Exception_CP14DTTrap,         // Trapped AArch32 LDC or STC access to CP14
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,           // Trapped access to SIMD or FP ID register
                       // Trapped BXJ instruction not supported in Armv8
                       Exception_PACTrap,            // Trapped invalid PAC use
                       Exception_CP14RRTrap,         // 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,        // PC alignment fault
                       Exception_DataAbort,          // Data Abort
                       Exception_PACFail,            // PAC Authentication failure
                       Exception_SPAlignment,        // SP alignment fault
                       Exception_FPTrappedException, // IEEE trapped FP exception
                       Exception_SError,             // SError interrupt
                       Exception_Breakpoint,         // (Hardware) Breakpoint
                       Exception_SoftwareStep,       // Software Step
                       Exception_Watchpoint,         // Watchpoint
                       Exception_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 exceptype,      // Exception class
                         bits(25)  syndrome,       // Syndrome record
                         bits(64)  vaddress,       // Virtual fault address
                         boolean   ipavalid,       // Physical fault address for second stage faults
is valid
                         bits(1)   NS,             // Physical fault address for second stage faults
is Non-secure or secure
                         bits(52)  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 exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();

    return r;
```

shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// ===============

ReservedValue()
    AArch64.UndefinedFault();
```

shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()
    AArch64.UndefinedFault();
```

I1.2.3   shared/functions

This section includes the following pseudocode functions:

shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)

    bits(6) result;
    case statuscode of
        when Fault_AddressSize       result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag        result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission        result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation       result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal      result = '010000';
        when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity        result = '011000';
        when Fault_SyncParityOnWalk  result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity       result = '011001';
        when Fault_AsyncExternal     result = '010001';
        when Fault_Alignment         result = '100001';
        when Fault_Debug             result = '100010';
        when Fault_TLBConflict       result = '110000';
        when Fault_HWUpdateAccessFlag result = '110001';
        when Fault_Lockdown          result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive         result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                    Unreachable();

    return result;
```

shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.s2fs1walk then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                          Fault_AddressSize};
    elsif fault.secondstage then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.
```

```
boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == 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 statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk,
Fault_SyncParityOnWalk,
                     Fault_AsyncExternal, Fault_AsyncParity });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);
```

shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE
otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk,
Fault_SyncParityOnWalk});

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);
```

shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor
```

```
boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;
```

shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != 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 - (HighestSetBit(x) + 1);
```

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/LowestSetBit

```
// LowestSetBit()
// =============

integer LowestSetBit(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/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 + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    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 + 2^i;
        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

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
FFmul09(in3<c*8+:8>);
        out1<c*8+:8> = FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> = FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
FFmul0B(in3<c*8+:8>);
        out3<c*8+:8> = FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
FFmul0E(in3<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 24+:8> : op< 48+:8> : op< 72+:8> : op< 96+:8> :
        op<120+:8> : op< 16+:8> : op< 40+:8> : op< 64+:8> :
```

```
                         op< 88+:8> : op<112+:8> : op<  8+:8> : op< 32+:8> :
                         op< 56+:8> : op< 80+:8> : op<104+:8> : op<  0+:8>
                    );
```

shared/functions/crypto/AESInvSubBytes

```
 // AESInvSubBytes()
 // ================
 // Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

 bits(128) AESInvSubBytes(bits(128) op)
     // Inverse S-box values
     bits(16*16*8) GF2_inv = (
         /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
         /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
         /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
         /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
         /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
         /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
         /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
         /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
         /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
         /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
         /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
         /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
         /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
         /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
         /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
         /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
         /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
     );
     bits(128) out;
     for i = 0 to 15
         out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
     return out;
```

shared/functions/crypto/AESMixColumns

```
 // AESMixColumns()
 // ===============
 // Transformation in the Cipher that takes all of the columns of the
 // State and mixes their data (independently of one another) to
 // produce new columns.

 bits(128) AESMixColumns(bits (128) op)
     bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
     bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
     bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
     bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

     bits(4*8) out0;
     bits(4*8) out1;
     bits(4*8) out2;
     bits(4*8) out3;

     for c = 0 to 3
         out0<c*8+:8> = FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR       in2<c*8+:8>  EOR
 in3<c*8+:8>;
         out1<c*8+:8> =          in0<c*8+:8>  EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
 in3<c*8+:8>;
         out2<c*8+:8> =          in0<c*8+:8>  EOR          in1<c*8+:8>  EOR FFmul02(in2<c*8+:8>) EOR
 FFmul03(in3<c*8+:8>);
         out3<c*8+:8> = FFmul03(in0<c*8+:8>) EOR          in1<c*8+:8>  EOR          in2<c*8+:8>  EOR
 FFmul02(in3<c*8+:8>);

     return (
```

```
            out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
            out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
            out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
            out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
        );
```

shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 88+:8> : op< 48+:8> : op<  8+:8> : op< 96+:8> :
        op< 56+:8> : op< 16+:8> : op<104+:8> : op< 64+:8> :
        op< 24+:8> : op<112+:8> : op< 72+:8> : op< 32+:8> :
        op<120+:8> : op< 80+:8> : op< 40+:8> : op<  0+:8>
    );
```

shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0        */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

shared/functions/crypto/FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0        */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
```

```
                /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
                /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
                /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
                /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
                /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
                /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
                /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
                /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
                /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
                /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
                /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
                /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
                /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
            );
            return FFmul_02<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
```

```
                /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
            );
            return FFmul_09<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*        F E D C B A 9 8 7 6 5 4 3 2 1 0        */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*        F E D C B A 9 8 7 6 5 4 3 2 1 0        */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*        F E D C B A 9 8 7 6 5 4 3 2 1 0        */
```

```
                          /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
                          /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
                          /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
                          /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
                          /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
                          /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
                          /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
                          /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
                          /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
                          /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
                          /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
                          /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
                          /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
                          /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
                          /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
                          /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
        );
        return FFmul_0E<UInt(b)*8+:8>;
```

shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// ============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// ====================
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
```

```
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

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/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring =
0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491ef987a33540b43edcfac62e4b31ca
9c908e89580df94fa758f3fa64707a7fcf37317ba83593c19e6854fa8686b81b27164da8bf8eb0f4b70569d351e240e5e6358d1a2
25227c3b01217887d40046579fd327524c3602e7a0c4c89eeabf8ad240c738b5a3f7f2cef96115a1e0ae5da49b341a55ad933230f
58cb1e31df6e22e8266ca60c02923ab0d534e6fd5db3745defd8e2f03ff6a726d6c5b518d1baf92bbddbc7f11d95c411f105ad80a
c13188a5cd7bbd2d74d012b8e5b4b08969974a0c96777e65b9f109c56ec68418f07dec3adc4d2079ee5f3ed7cb3948<2047:0>;
```

```
        sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
        return sboxout;
```

shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================

// Clear the local Exclusives 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 Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ===================

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support";
```

shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support";
```

shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ==============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// =============
// Returns TRUE if support for changing block size without requring break-before-make is implemented.

boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveDGHExt

```
// HaveDGHExt()
// ===========
// Returns TRUE if Data Gathering Hint instruction support is implemented, and FALSE otherwise.
```

```
boolean HaveDGHExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";
```

shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// ============

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has
Dot Product extension");
```

shared/functions/extension/HaveDoPD

```
// HaveDoPD()
// ==========
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";
```

shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";
```

shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDExt()
    return HasArchVersion(ARMv8p5);
```

shared/functions/extension/HaveEL1VMSAExt

```
// HaveEL1VMSAExt()
// ================
// Returns TRUE if VMSA is supported at stage1 EL1&0 translation regime, FALSE otherwise.

boolean HaveEL1VMSAExt()
    return ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0010';
```

shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()
// ======================

boolean HaveExtendedCacheSets()
    return HasArchVersion(ARMv8p3);
```

shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// ==========================

boolean HaveExtendedECDebugEvents()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// ============================

boolean HaveExtendedExecuteNeverExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()
// =============

boolean HaveFCADDExt()
    return HasArchVersion(ARMv8p3);
```

shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ===============

boolean HaveFJCVTZSExt()
    return HasArchVersion(ARMv8p3);
```

shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// ===============================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext() then return FALSE;
    if HasArchVersion(ARMv8p4) then return TRUE;
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// ======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// ============

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()
// ===========
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveIESB

```
// HaveIESB()
// ==========

boolean HaveIESB()
    return (HaveRASExt() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

shared/functions/extension/HaveLSE2Ext

```
// HaveLSE2Ext()
// ============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// =========================
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// ================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// ==============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// ========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals is implemented.
```

```
boolean HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// ====================

boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveSmallPageTblExt

```
// HaveSmallPageTblExt()
// =====================
// Returns TRUE if Small Page Table Support is implemented.

boolean HaveSmallPageTblExt()
    return HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Page Table extension";
```

shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// ==========================
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is
implemented.

boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);
```

shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/HaveV82Debug

```
// HaveV82Debug()
// ==============

boolean HaveV82Debug()
    return HasArchVersion(ARMv8p2);
```

shared/functions/extension/InsertIESBBeforeException

```
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.
boolean InsertIESBBeforeException(bits(2) el);
```

shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};

    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 {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1  = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
```

```
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);

    return result;
```

shared/functions/float/fpcommon/IsDenormalizedValue

```
// IsDenormalizedValue()
// =====================
// Checks either a single-precision or a double-precision floating-point
// value is denormalized.

boolean IsDenormalizedValue(bits(N) fpval)
    assert N IN {32,64};

    case N of
        when 32
            exp32    = fpval<30:23>;
            frac32   = fpval<22:0>;
            isDenormal = IsZero(exp32) && !IsZero(frac32);
        when 64
            exp64    = fpval<62:52>;
            frac64   = fpval<51:0>;
            isDenormal = IsZero(exp64) && !IsZero(frac64);

    return isDenormal;
```

shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = '0011';
        if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';

    return result;
```

shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
```

```
        (type1,sign1,value1) = FPUnpack(op1, fpcr);
        (type2,sign2,value2) = FPUnpack(op2, fpcr);
        op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
        op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

        if op1_nan || op2_nan 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, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);

    return result;
```

shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);

    return result;
```

shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
```

```
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);
    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType 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/FPCRType

```
type FPCRType;
```

shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF;  // P
        when '11' result = FPRounding_NEGINF;  // M

    return result;
```

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';

    bits(E) exp  = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;
```

shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
```

```
                      (type1,sign1,value1) = FPUnpack(op1, fpcr);
                      (type2,sign2,value2) = FPUnpack(op2, fpcr);
                      (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

                 if !done then
                      inf1 = (type1 == FPType_Infinity);
                      inf2 = (type2 == FPType_Infinity);
                      zero1 = (type1 == FPType_Zero);
                      zero2 = (type2 == FPType_Zero);
                      invalidop = (inf1 && inf2) || (zero1 && zero2);
                      if invalidop 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);
    bits(E) exp  = Ones(E);
    bits(F) frac = Zeros(F);

    return sign : exp : frac;
```

shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
```

```
                         // for a denormalized number even though the result is exact.
                         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 {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');

    result = FPMax(op1, op2, fpcr);

    return result;
```

shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
```

```
                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 {16,32,64};
        (type1,-,-) = FPUnpack(op1, fpcr);
        (type2,-,-) = FPUnpack(op2, fpcr);

        // Treat a single quiet-NaN as +Infinity
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('0');
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('0');

        result = FPMin(op1, op2, fpcr);

        return result;
```

shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)

        assert N IN {16,32,64};
        (type1,sign1,value1) = FPUnpack(op1, fpcr);
        (type2,sign2,value2) = FPUnpack(op2, fpcr);
        (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
        if !done then
            inf1 = (type1 == FPType_Infinity);
            inf2 = (type2 == FPType_Infinity);
            zero1 = (type1 == FPType_Zero);
            zero2 = (type2 == FPType_Zero);
            invalidop = (inf1 && zero2) || (zero1 && inf2);
            if invalidop 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. The 'fpcr' argument
// supplies the FPCR control bits.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
        assert N IN {16,32,64};

        rounding = FPRoundingMode(fpcr);
```

```
                (typeA,signA,valueA) = FPUnpack(addend, fpcr);
                (type1,sign1,value1) = FPUnpack(op1, fpcr);
                (type2,sign2,value2) = FPUnpack(op2, fpcr);
                inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
                inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

                (done,result) = FPProcessNaNs3(typeA, type1, type2,
                                               addend, op1, op2, fpcr);

                if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
                    result = FPDefaultNaN();
                    FPProcessException(FPExc_InvalidOp, fpcr);

                if !done then
                    infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

                    // Determine sign and type product will have if it does not cause an
                    // Invalid Operation.
                    signP = sign1 EOR sign2;
                    infP  = inf1 || inf2;
                    zeroP = zero1 || zero2;

                    // Non SNaN-generated Invalid Operation cases are multiplies of zero
                    // by infinity and additions of opposite-signed infinities.
                    invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

                    if invalidop then
                        result = FPDefaultNaN();
                        FPProcessException(FPExc_InvalidOp, fpcr);
                    // Other cases involving infinities produce an infinity of the same sign.
                    elsif (infA && signA == '0') || (infP && signP == '0') then
                        result = FPInfinity('0');
                    elsif (infA && signA == '1') || (infP && signP == '1') then
                        result = FPInfinity('1');

                    // Cases where the result is exactly zero and its sign is not determined by the
                    // rounding mode are additions of same-signed zeros.
                    elsif zeroA && zeroP && signA == signP then
                        result = FPZero(signA);

                    // Otherwise calculate numerical result and round it.
                    else
                        result_value = valueA + (value1 * value2);
                        if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                            result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                            result = FPZero(result_sign);
                        else
                            result = FPRound(result_value, fpcr);

                return result;
```

shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
```

```
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        if !done then
            infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

            // Determine sign and type product will have if it does not cause an
            // Invalid Operation.
            signP = sign1 EOR sign2;
            infP = inf1 || inf2;
            zeroP = zero1 || zero2;

            // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
            // additions of opposite-signed infinities.
            invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

            if invalidop then
                result = FPDefaultNaN();
                FPProcessException(FPExc_InvalidOp, fpcr);

            // Other cases involving infinities produce an infinity of the same sign.
            elsif (infA && signA == '0') || (infP && signP == '0') then
                result = FPInfinity('0');
            elsif (infA && signA == '1') || (infP && signP == '1') then
                result = FPInfinity('1');

            // Cases where the result is exactly zero and its sign is not determined by the
            // rounding mode are additions of same-signed zeros.
            elsif zeroA && zeroP && signA == signP then
                result = FPZero(signA);

            // Otherwise calculate numerical result and round it.
            else
                result_value = valueA + (value1 * value2);
                if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                    result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                    result = FPZero(result_sign);
                else
                    result = FPRound(result_value, fpcr);

    return result;
```

shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                   FPCRType fpcr)

    assert N IN {32,64};

    bits(N) result;
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
```

```
        else
            done = FALSE; result = Zeros(); // 'Don't care' result

        return (done, result);
```

shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);

    return result;
```

shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};

    return NOT(op<N-1>) : op<N-2:0>;
```

shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)

    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp    cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow     cumul = 2;
        when FPExc_Underflow    cumul = 3;
        when FPExc_Inexact      cumul = 4;
        when FPExc_InputDenorm  cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrappedException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';

    return;
```

shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        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 {16,32,64};
            if type1 == FPType_SNaN then
                done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
            elsif type2 == FPType_SNaN then
                done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
            elsif type1 == FPType_QNaN then
                done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
            elsif type2 == FPType_QNaN then
                done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
            else
                done = FALSE;  result = Zeros();  // 'Don't care' result

            return (done, result);
```

shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

                        (boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                                          bits(N) op1, bits(N) op2, bits(N) op3,
                                                          FPCRType fpcr)
            assert N IN {16,32,64};

            if type1 == FPType_SNaN then
                done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
            elsif type2 == FPType_SNaN then
                done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
            elsif type3 == FPType_SNaN then
                done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
            elsif type1 == FPType_QNaN then
                done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
            elsif type2 == FPType_QNaN then
                done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
            elsif type3 == FPType_QNaN then
                done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
            else
                done = FALSE;  result = Zeros();  // 'Don't care' result

            return (done, result);
```

shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)

            assert N IN {16,32,64};
            (fptype,sign,value) = FPUnpack(operand, fpcr);

            if fptype == FPType_SNaN || fptype == FPType_QNaN then
                result = FPProcessNaN(fptype, operand, fpcr);
            elsif fptype == FPType_Infinity then
                result = FPZero(sign);
            elsif fptype == FPType_Zero then
```

```
                        result = FPInfinity(sign);
                        FPProcessException(FPExc_DivideByZero, fpcr);
                elsif (
                        (N == 16 && Abs(value) < 2.0^-16) ||
                        (N == 32 && Abs(value) < 2.0^-128) ||
                        (N == 64 && Abs(value) < 2.0^-1024)
                      ) then
                    case FPRoundingMode(fpcr) of
                        when FPRounding_TIEEVEN
                            overflow_to_inf = TRUE;
                        when FPRounding_POSINF
                            overflow_to_inf = (sign == '0');
                        when FPRounding_NEGINF
                            overflow_to_inf = (sign == '1');
                        when FPRounding_ZERO
                            overflow_to_inf = FALSE;
                    result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
                    FPProcessException(FPExc_Overflow, fpcr);
                    FPProcessException(FPExc_Inexact, fpcr);
                elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
                        && (
                            (N == 16 && Abs(value) >= 2.0^14) ||
                            (N == 32 && Abs(value) >= 2.0^126) ||
                            (N == 64 && Abs(value) >= 2.0^1022)
                          ) then
                    // Result flushed to zero of correct sign
                    result = FPZero(sign);

                    // Flush-to-zero never generates a trapped exception.
                    FPSR.UFC = '1';
                else
                    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
                    // calculate result exponent. Scaled value has copied sign bit,
                    // exponent = 1022 = double-precision biased version of -1,
                    // fraction = original fraction
                    case N of
                        when 16
                            fraction = operand<9:0> : Zeros(42);
                            exp = UInt(operand<14:10>);
                        when 32
                            fraction = operand<22:0> : Zeros(29);
                            exp = UInt(operand<30:23>);
                        when 64
                            fraction = operand<51:0>;
                            exp = UInt(operand<62:52>);

                    if exp == 0 then
                        if fraction<51> == '0' then
                            exp = -1;
                            fraction = fraction<49:0>:'00';
                        else
                            fraction = fraction<50:0>:'0';

                    integer scaled = UInt('1':fraction<51:44>);

                    case N of
                        when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
                        when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
                        when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

                    // scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
                    estimate = RecipEstimate(scaled);

                    // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
                    // Convert to scaled floating point result with copied sign bit,
                    // high-order bits from estimate, and exponent calculated above.

                    fraction = estimate<7:0> : Zeros(44);
```

```
                    if result_exp == 0 then
                        fraction = '1' : fraction<51:1>;
                    elsif result_exp == -1 then
                        fraction = '01' : fraction<51:2>;
                        result_exp = 0;

                    case N of
                        when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
                        when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
                        when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

            return result;
```

shared/functions/float/fprecipestimate/RecipEstimate

```
// Compute estimate of reciprocal of 9-bit fixed-point number
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
    assert 256 <= a && a < 512;
    a = a*2+1; // round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};

    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;

    bits(N)         result;
    bits(esize)     exp;
    bits(esize)     max_exp;
    bits(N-(esize+1)) frac = Zeros();

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, 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()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPRoundBase(op, fpcr, rounding);

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

shared/functions/float/fpround/FPRoundBase

```
// FPRoundBase()
// =============
// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    if (((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
```

```
                        FPProcessException(FPExc_Underflow, fpcr);

            // Round result according to rounding mode.
            case rounding of
                when FPRounding_TIEEVEN
                    round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                    overflow_to_inf = TRUE;
                when FPRounding_POSINF
                    round_up = (error != 0.0 && sign == '0');
                    overflow_to_inf = (sign == '0');
                when FPRounding_NEGINF
                    round_up = (error != 0.0 && sign == '1');
                    overflow_to_inf = (sign == '1');
                when FPRounding_ZERO, FPRounding_ODD
                    round_up = FALSE;
                    overflow_to_inf = FALSE;

            if round_up then
                int_mant = int_mant + 1;
                if int_mant == 2^F then      // Rounded up from denormalized to normalized
                    biased_exp = 1;
                if int_mant == 2^(F+1) then  // Rounded up to next exponent
                    biased_exp = biased_exp + 1;   int_mant = int_mant DIV 2;

            // Handle rounding to odd aka Von Neumann rounding
            if error != 0.0 && rounding == FPRounding_ODD then
                int_mant<0> = '1';

            // Deal with overflow and generate result.
            if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
                if biased_exp >= 2^E - 1 then
                    result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
                    FPProcessException(FPExc_Overflow, fpcr);
                    error = 1.0;  // Ensure that an Inexact exception occurs
                else
                    result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
            else                                  // Alternative half precision
                if biased_exp >= 2^E then
                    result = sign : Ones(N-1);
                    FPProcessException(FPExc_InvalidOp, fpcr);
                    error = 0.0;  // Ensure that an Inexact exception does not occur
                else
                    result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

            // Deal with Inexact exception.
            if error != 0.0 then
                FPProcessException(FPExc_Inexact, fpcr);

            return result;
```

shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    return FPRoundBase(op, fpcr, rounding);
```

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 in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)

    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        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/fproundintn/FPRoundIntN

```
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == 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;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign);
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO);
```

```
            // Generate inexact exceptions.
            if error != 0.0 then
                FPProcessException(FPExc_Inexact, fpcr);

        return result;
```

shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(operand, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);

        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0]
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros( 2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
```

```
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

        return result;
```

shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1;      // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1;   // discard bottom bit
        a = (a+1)*2;  // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a) do
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);

    return result;
```

shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
```

```
                  invalidop =  inf1 && inf2 && sign1 == sign2;

                  if invalidop then
                      result = FPDefaultNaN();
                      FPProcessException(FPExc_InvalidOp, fpcr);
                  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
                      result = FPInfinity('0');
                  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
                      result = FPInfinity('1');
                  elsif zero1 && zero2 && sign1 == NOT(sign2) then
                      result = FPZero(sign1);
                  else
                      result_value = value1 - value2;
                      if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                          result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                          result = FPZero(result_sign);
                      else
                          result = FPRound(result_value, fpcr, rounding);

        return result;
```

shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    case rounding of
```

```
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';
```

```
        if fptype == FPType_Infinity then result = 0;

        return (result<N-1:0>, Z);
```

shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

shared/functions/float/fptype/FPType

```
enumeration FPType        {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                           FPType_QNaN, FPType_SNaN};
```

shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero;   value = 0.0;
            else
```

```
                    fptype = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
                elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
                    if IsZero(frac16) then
                        fptype = FPType_Infinity;  value = 2.0^1000000;
                    else
                        fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                        value = 0.0;
                else
                    fptype = FPType_Nonzero;
                    value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

        elsif N == 32 then

            sign   = fpval<31>;
            exp32  = fpval<30:23>;
            frac32 = fpval<22:0>;
            if IsZero(exp32) then
                // Produce zero if value is zero or flush-to-zero is selected.
                if IsZero(frac32) || fpcr.FZ == '1' then
                    fptype = FPType_Zero;  value = 0.0;
                    if !IsZero(frac32) then  // Denormalized input flushed to zero
                        FPProcessException(FPExc_InputDenorm, fpcr);
                else
                    fptype = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
            elsif IsOnes(exp32) then
                if IsZero(frac32) then
                    fptype = FPType_Infinity;  value = 2.0^1000000;
                else
                    fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                    value = 0.0;
            else
                fptype = FPType_Nonzero;
                value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

        else // N == 64

            sign   = fpval<63>;
            exp64  = fpval<62:52>;
            frac64 = fpval<51:0>;
            if IsZero(exp64) then
                // Produce zero if value is zero or flush-to-zero is selected.
                if IsZero(frac64) || fpcr.FZ == '1' then
                    fptype = FPType_Zero;  value = 0.0;
                    if !IsZero(frac64) then  // Denormalized input flushed to zero
                        FPProcessException(FPExc_InputDenorm, fpcr);
                else
                    fptype = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
            elsif IsOnes(exp64) then
                if IsZero(frac64) then
                    fptype = FPType_Infinity;  value = 2.0^1000000;
                else
                    fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                    value = 0.0;
            else
                fptype = FPType_Nonzero;
                value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

        if sign == '1' then value = -value;

        return (fptype, sign, value);
```

shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP <-> FP conversion instructions.
```

```
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

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/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
```

```
        if msbit == 63 then
            return vaddress;
        elsif PSTATE.EL IN {EL0, EL1} && vaddress<msbit> == '1' then
            return SignExtend(vaddress<msbit:0>);
        else
            return ZeroExtend(vaddress<msbit:0>);
```

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_ORDEREDATOMIC,              // Load-Acquire and Store-Release with atomic
access
                     AccType_ORDEREDATOMICRW,
                     AccType_UNPRIV,                     // Load and store unprivileged
                     AccType_IFETCH,                     // Instruction fetch
                     AccType_PTW,                        // Page table walk
                     // Other operations
                     AccType_DC,                         // Data cache maintenance
                     AccType_DC_UNPRIV,                  // Data cache maintenance instruction used at
EL0
                     AccType_IC,                         // Instruction cache maintenance
                     AccType_DCZVA,                      // DC ZVA instructions
                     AccType_AT};                        // Address translation
```

shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    AccType acctype,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fs1walk,
    integer level
)
```

shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```

shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord     fault,    // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress     paddress,
    bits(64)        vaddress
)
```

shared/functions/memory/Allocation

```
constant bits(2) MemHint_No  = '00';    // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA  = '01';    // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA  = '10';    // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

shared/functions/memory/BigEndian

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

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/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/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

shared/functions/memory/CreateAccessDescriptorPTW

```
// CreateAccessDescriptorPTW()
// ===========================

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
                                           boolean s2fs1walk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = TRUE;
    accdesc.s2fs1walk = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
```

shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types, boolean vmid_sensitive);
```

shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean vmid_sensitive);
```

shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
    boolean AF,                 // AF needs to be set
    boolean AP,                 // AP[2] / S2AP[2] will be modified
    AddressDescriptor descaddr  // Descriptor to be updated
)
```

shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// =============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            tbi = TCR_EL2.TBI;
            if HavePACExt() then tbid = TCR_EL2.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
```

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_HWUpdateAccessFlag,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    statuscode, // Fault Status
                     AccType   acctype,    // Type of access that faulted
                     FullAddress ipaddress, // Intermediate physical address
                     boolean   s2fs1walk,  // Is on a Stage 1 page table walk
                     boolean   write,      // TRUE for a write, FALSE for a read
                     integer   level,      // For translation, access flag and permission faults
                     bit       extflag,    // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean   secondstage, // Is a Stage 2 abort
                     bits(4)   domain,     // Domain number, AArch32 only
                     bits(2)   errortype,  // [Armv8.2 RAS] AArch32 AET or AArch64 SET
                     bits(4)   debugmoe)   // Debug method of entry, from AArch32 only
```

shared/functions/memory/FullAddress

```
type FullAddress is (
    bits(52) address,
    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/IsStage1VMSA

```
// IsStage1VMSA()
// ==============
// Return TRUE if stage1 EL1&0 translation regime is VMSA and returns FALSE if PMSA.

boolean IsStage1VMSA()
    if HaveEL1VMSAExt() && S1TranslationRegime() == EL1 then
        return VTCR_EL2.MSA == '1';
    else
        return FALSE;
```

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/MPURecord

```
type MPURecord is (
    bit              CnP,            // [Armv8.2] TLB entry can be shared between different PEs
    Permissions      perms,
    AddressDescriptor addrdesc
)
```

shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType      memtype,

    DeviceType   device,      // For Device memory types
    MemAttrHints inner,       // Inner hints and attributes
    MemAttrHints outer,       // Outer hints and attributes
    boolean      is_fwb,
    boolean      shareable,
    boolean      outershareable
)
```

shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     xxn,  // [Armv8.2] Extended execute-never bit for stage 2
    bit     pxn   // Privileged execute-never bit
)
```

shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
SpeculativeStoreBypassBarrierToPA();
```

shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
SpeculativeStoreBypassBarrierToVA();
```

shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions      perms,
    bit              nG,          // '0' = Global, '1' = not Global
    bits(4)          domain,      // AArch32 only
    boolean          contiguous,  // Contiguous bit from page table
    integer          level,       // AArch32 Short-descriptor format: Indicates Section/Page
```

```
    integer         blocksize,      // Describes size of memory translated in KBytes
    DescriptorUpdate descupdate,     // [Armv8.1] Context for h/w update of table descriptor
    bit             CnP,            // [Armv8.2] TLB entry can be shared between different PEs
    AddressDescriptor addrdesc
)
```

shared/functions/memory/_Mem

```
// These two _Mem[] accessors are the hardware operations which perform single-copy atomic,
// aligned, little-endian memory accesses of size bytes from/to the underlying physical
// memory array of bytes.
//
// The functions address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc];

_Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;
```

shared/functions/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
```

shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============

// Set program counter to a new address, with a branch type
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
```

```
                     BranchType_DBGEXIT,      // Exit from Debug state
                     BranchType_RET,          // Indirect branch with function return hint
                     BranchType_DIR,          // Direct branch
                     BranchType_INDIR,        // Indirect branch
                     BranchType_EXCEPTION,    // Exception entry
                     BranchType_RESET,        // Reset
                     BranchType_UNKNOWN};     // Other
```

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 sequentially 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;
    case PSTATE.EL of
        when EL1         result = SPSR_EL1;
        when EL2         result = SPSR_EL2;
        otherwise        Unreachable();
    return result;

// SPSR[] - assignment form
// ========================
```

```
        SPSR[] = bits(32) value
            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
    , ARMv8p2
    , ARMv8p3
    , ARMv8p4
    , ARMv8p5
    , ARMv9p0
};
```

shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE

ClearEventRegister()
    EventRegister = '0';
    return;
```

shared/functions/system/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt
ClearPendingPhysicalSError();
```

shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt
ClearPendingVirtualSError();
```

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/ConsumptionOfSpeculativeDataBarrier

```
ConsumptionOfSpeculativeDataBarrier();
```

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/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

shared/functions/system/EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with SCR_EL3.NS==1 when Non-secure EL2 is implemented, or
// - with SCR_EL3.NS==0 when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return TRUE;
```

shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then                    // AArch64 state
        el = spsr<3:2>;
        if HighestELUsingAArch32() then        // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                 // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then              // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then // for EL0, M[0] must be 0
            valid = FALSE;
        else
            valid = TRUE;
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ===============

boolean ELUsingAArch32(bits(2) el)
    return FALSE;
```

shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// ================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return (TRUE, FALSE);
```

shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state
EnterLowPowerState();
```

shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

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<20>    = PSTATE.IL;
    if HaveDITExt() then spsr<24> = PSTATE.DIT;
    if HaveUAOExt() then spsr<23> = PSTATE.UAO;
    spsr<21>    = PSTATE.SS;
    if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
    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) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                    // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;                    // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;                     // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                    // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;                     // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return FALSE;
```

shared/functions/system/HaveAnyAArch64

```
// HaveAnyAArch64()
// ================
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();
```

shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL2,EL1,EL0} then
        return TRUE;                         // EL2, EL1 and EL0 must exist
    else
        return FALSE;
```

shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
```

```
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// ============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

shared/functions/system/HighestEL

```
// HighestEL()
// ==========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

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_DGH

```
// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
```

shared/functions/system/Hint_Yield

```
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();
```

shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =======================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for illegal return:
    //    * To an unimplemented Exception level.
    //    * To EL2 in Secure state, when SecureEL2 is not enabled.
    //    * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //    * To AArch64 state with SPSR.M[1]==1.
    //    * To AArch32 state with an illegal value of SPSR.M.
```

```
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier(boolean vmid_sensitive);
```

shared/functions/system/InterruptPending

```
// InterruptPending()
// ==================
// Return TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    pending_physical_interrupt = (IRQPending() || FIQPending() ||
                                  IsPhysicalSErrorPending());
    pending_virtual_interrupt  = ((HCR_EL2.<VSE,VI,VF> AND
                                   HCR_EL2.<AMO,IMO,FMO>) != '000');
    return pending_physical_interrupt || pending_virtual_interrupt;
```

shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

shared/functions/system/IsHighestEL

```
// IsHighestEL()
// ============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

shared/functions/system/IsPhysicalSErrorPending

```
// Return TRUE if a physical SError interrupt is pending
boolean IsPhysicalSErrorPending();
```

shared/functions/system/IsSecure

```
// IsSecure()
// ==========
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    return TRUE;
```

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()
    return TRUE;
```

shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    return TRUE;
```

shared/functions/system/IsSynchronizablePhysicalSErrorPending

```
// Return TRUE if a synchronizable physical SError interrupt is pending
boolean IsSynchronizablePhysicalSErrorPending();
```

shared/functions/system/IsVirtualSErrorPending

```
// Return TRUE if a virtual SError interrupt is pending
boolean IsVirtualSErrorPending();
```

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,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit       [v8.1]
    bits (1) UAO,      // User Access Override              [v8.2]
    bits (1) DIT,      // Data Independent Timing           [v8.4]
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception Level
    bits (1) nRW,      // not Register Width: 0=64, 1=32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag        [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags       [AArch32 only]
    bits (1) SSBS,     // Speculative Store Bypass Safe
    bits (8) IT,       // If-then bits, RES0 in CPSR        [AArch32 only]
    bits (1) J,        // J bit, RES0                       [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR             [AArch32 only]
    bits (1) E,        // Endianness bit                    [AArch32 only]
    bits (5) M         // Mode field                        [AArch32 only]
)
```

shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();
```

shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed

SendEventLocal()
    EventRegister = '1';
    return;
```

shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
```

```
                PSTATE.IL = spsr<20>;
                PSTATE.nRW = '0';
                PSTATE.EL  = spsr<3:2>;
                PSTATE.SP  = spsr<0>;
                if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;

        // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
        // copied from SPSR.
        if PSTATE.IL == '1' && PSTATE.nRW == '1' then
            if ConstrainUnpredictableBool() then spsr<5> = '0';

        // State that is reinstated regardless of illegal exception return
        PSTATE.<N,Z,C,V> = spsr<31:28>;
        if HavePANExt() then PSTATE.PAN = spsr<22>;
        PSTATE.<D,A,I,F> = spsr<9:6>;
        return;
```

shared/functions/system/SpeculationBarrier

```
 SpeculationBarrier();
```

shared/functions/system/SynchronizeContext

```
 SynchronizeContext();
```

shared/functions/system/SynchronizeErrors

```
 // Implements the error synchronization event.
 SynchronizeErrors();
```

shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
 // Take any pending unmasked physical SError interrupt
 TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

shared/functions/system/TakeUnmaskedSErrorInterrupts

```
 // Take any pending unmasked physical SError interrupt or unmasked virtual SError
 // interrupt.
 TakeUnmaskedSErrorInterrupts();
```

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()
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed

WaitForEvent()
    if EventRegister == '0' then
        EnterLowPowerState();
    return;
```

shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset

WaitForInterrupt()
    EnterLowPowerState();
    return;
```

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,           // Instruction executes with
                                                     //   no change or side-effect to its described
behavior
                          Constraint_UNKNOWN,        // Destination register has UNKNOWN value
                          Constraint_UNDEF,          // Instruction is UNDEFINED
                          Constraint_UNDEFEL0,       // Instruction is UNDEFINED at EL0 only
                          Constraint_NOP,            // Instruction executes as NOP
                          Constraint_TRUE,
                          Constraint_FALSE,
                          Constraint_DISABLED,
                          Constraint_UNCOND,         // Instruction executes unconditionally
                          Constraint_COND,           // Instruction executes conditionally
                          Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
                          // Load-store
                          Constraint_WBSUPPRESS, Constraint_FAULT,
                           Constraint_MPU_FAULT,     // Raise MPU fault
                           Constraint_MPU_ATTR_UNKNOWN, // MPU Attribute is UNKNOWN
                          // IPA too large
                          Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

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;
    elsif 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(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)

        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N IN {16,32};
```

```
                    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
                        result = Ones(N);
                    else
                        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

                        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
                        case N of
                            when 16 estimate = RecipEstimate(UInt(operand<15:7>));
                            when 32 estimate = RecipEstimate(UInt(operand<31:23>));

                        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
                        result = estimate<8:0> : Zeros(N-9);

                    return result;
```

shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

I1.2.4   shared/translation

This section includes the following pseudocode functions:

shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// ======================
// Combines cacheability attributes and allocation hints from stage 1 and stage 2

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc, AccType s2acctype)
```

```
                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

                if result.attrs == MemAttr_NC then
                    result.hints = MemHint_No;
                else
                    result.hints = s1desc.hints;
                result.transient = s1desc.transient;

                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;
                elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
                    result = DeviceType_nGnRE;
                elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
                    result = DeviceType_nGRE;
                else
                    result = DeviceType_GRE;

                return result;
```

shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// ======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
        assert !IsZero(attrfield);
        MemAttrHints result;
        if S1CacheDisabled(acctype) then               // Force Non-cacheable
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
        else
            if attrfield<3:2> == '00' then          // Write-through transient
                result.attrs = MemAttr_WT;
                result.hints = attrfield<1:0>;
                result.transient = TRUE;
            elsif attrfield<3:0> == '0100' then     // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
                result.transient = FALSE;
            elsif attrfield<3:2> == '01' then       // Write-back transient
                result.attrs = MemAttr_WB;
                result.hints = attrfield<1:0>;
                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.memtype == 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)
    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;

    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';

    // Device memory
    if (apply_force_writeback && attr<2> == '0') || attr<3:2> == '00' then
        memattrs.memtype = 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;

    // Normal memory
    elsif apply_force_writeback then
        if attr<2> == '1' then
            memattrs.memtype = MemType_Normal;
            memattrs.inner.attrs = attr<1:0>;
            memattrs.outer.attrs = attr<1:0>;
            memattrs.shareable = SH<1> == '1';
            memattrs.outershareable = SH == '10';
    elsif attr<1:0> != '00' then
```

```
                        memattrs.memtype = 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)
        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 attr != '00';

        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/S2MPUAttrDecode

```
// S2MPUAttrDecode()
// =================
// Derives stage 2 MPU attributes, using the MAIR, to orthogonal attributes and hints.

MemoryAttributes S2MPUAttrDecode(bits(2) SH, bits(3) attrIndx, AccType acctype)

        MemoryAttributes memattrs;
        apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
        memattrs.is_fwb = FALSE;
        index = 8 * UInt(attrIndx);
        attrfield = MAIR_EL2<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 memory
                    memattrs.memtype = 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 apply_force_writeback then
                    if attrfield<7:4> != attrfield<3:0> then
                        memattrs.inner =  MemAttrHints UNKNOWN;
                        memattrs.outer =  MemAttrHints UNKNOWN;
                        memattrs.shareable = boolean UNKNOWN;
                        memattrs.outershareable = boolean UNKNOWN;
                    else
                        // Check whether FWB should be applied
                        memattrs.is_fwb = if attrfield<7:6> == '11' then TRUE else FALSE;
                        // If FWB is enabled, then cacheability control is applied after applying FWB in
                        // CombineS1S2Desc(). Else cacheability is applied in S2MPUConvertAttrsHints()
                        if !memattrs.is_fwb then
                            memattrs.memtype = MemType_Normal;
                            memattrs.inner = S2MPUConvertAttrsHints(attrfield<3:0>, acctype);
                            memattrs.outer = S2MPUConvertAttrsHints(attrfield<7:4>, acctype);
                            memattrs.shareable = SH<1> == '1';
                            memattrs.outershareable = SH == '10';
                else
                    memattrs.memtype = MemType_Normal;
                    memattrs.inner = S2MPUConvertAttrsHints(attrfield<3:0>, acctype);
                    memattrs.outer = S2MPUConvertAttrsHints(attrfield<7:4>, acctype);
                    memattrs.shareable = SH<1> == '1';
                    memattrs.outershareable = SH == '10';

                return MemAttrDefaults(memattrs);
```

shared/translation/attrs/S2MPUConvertAttrsHints

```
// S2MPUConvertAttrsHints()
// ========================
// Converts the attribute fields for Normal memory as used in stage 2 MPU to
// orthogonal attributes and hints

MemAttrHints S2MPUConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);

    MemAttrHints result;
    if S2CacheDisabled(acctype) then          // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;

    else
        if attrfield<3:2> == '00' then         // Write-through
            result.attrs = MemAttr_WT;
            result.hints = MemHint_RWA;

        elsif attrfield<3:0> == '0100' then    // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;

        elsif attrfield<3:2> == '01' || attrfield<3:2> == '11' then   // Write-back
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RWA;
        else                                   // Write-through
```

```
                            result.attrs = MemAttr_WT;
                            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, boolean secondstage)

    MemAttrHints result;

    if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
        // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                   // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10'                   // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                   // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes memattrs;

    AccType acctype = AccType_NORMAL;

    memattrs.memtype = MemType_Normal;
    memattrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    memattrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    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 (EL2Enabled() && 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()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

shared/translation/translation/VAMax

```
// VAMax()
// ======
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

shared/translation/validation/HighestELAccess

```
// HighestELAccess()
// =================
// Returns TRUE if the access is from the highest EL in the translation regime, otherwise
// FALSE.

boolean HighestELAccess(boolean secondstage)
    if secondstage then return FALSE;
    elsif S1TranslationRegime() == PSTATE.EL then return TRUE;
    else return FALSE;
```

I1.2.5 See also

In the Arm Architecture Reference Manual

- Pseudocode for AArch64 operation.
- Pseudocode description of debug exceptions.
- Pseudocode description of general memory System instructions.
- Appendix K13 *Arm Pseudocode Definition*.

Glossary

A64 instruction   A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned.

Advanced SIMD   A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

Architecturally mapped

Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

Architecturally UNKNOWN

An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also IMPLEMENTATION DEFINED.

CONSTRAINED UNPREDICTABLE

Where an instruction can result in UNPREDICTABLE behavior, the Armv8 architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Execution at Non-secure EL1 or EL0 of an instruction that is CONSTRAINED UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.

Armv8-R AArch64

Architecture described in this supplement.

EL1 MPU   Memory Protection Unit that can be configured from EL1 or EL2. EL1 MPU is used by software running at EL1.

EL2 MPU   Memory Protection Unit that can be configured only from EL2. EL2 MPU is used by software running at EL2.

Flat address mapping

Is where the physical address for every access is equal to its virtual address.

IMPLEMENTATION DEFINED

Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

Intermediate physical address (IPA)

An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical address (PA), Virtual address (VA).

Load/Store architecture

An architecture where data-processing operations only operate on register contents, not directly on memory contents.

Memory Protection Unit (MPU)

A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

MPU   *See* Memory Protection Unit (MPU).

PA   *See* Physical address (PA).

PE   *See* Processing element (PE).

Physical address (PA)

An address that identifies a location in the physical memory map.

See also Intermediate physical address (IPA), Virtual address (VA).

PMSA

Protected Memory System Architecture - implementing an MPU

Processing element (PE)

The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Protection region

A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit   *See* Memory Protection Unit (MPU).

RES0   A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:
- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

—— **Note** ——
- RES0 is not used in descriptions of instruction encodings.

- Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.

2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.
     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.
  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES1   A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

• Is RES1 in some defined architectural context.

• Has different defined behavior in a different architectural context.

———— **Note** ————

• RES1 is not used in descriptions of instruction encodings.

• Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

———————————

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
    • Reads of the bit always return 1.
    • Writes to the bit are ignored.

2. The bit can be written. In this case:
    • An indirect write to the register sets the bit to 1.
    • A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.
      If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
    • A direct write to the bit must update a storage location associated with the bit.
    • The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

• An indirect write to the register sets the bit to 1.

• A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

    ———— **Note** ————

    As indicated in this list, this value might be written by an indirect write to the register.

    ———————————

    If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

• A direct write to the bit must update a storage location associated with the bit.

• While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

• Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.

• The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

RAZ   *See* Read-As-Zero (RAZ).

Read-As-Zero (RAZ)

Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBZP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

Read-As-One (RAO)

Hardware must implement the field as reading as all 1s.

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

SBO   *See* Should-Be-One (SBO).

SBOP   *See* Should-Be-One-or-Preserved (SBOP).

Should-Be-Zero-or-Preserved (SBZP)

From the introduction of the Armv8 architecture, the description *Should-Be-Zero-or-Preserved (SBZP)* is superseded by *RES0*.

------ **Note** ------

The Armv7 Large Physical Address Extension modified the definition of SBZP for register bits that are SBZP in some but not all contexts. The behavior of these bits is covered by the *RES0* definition, but not by the generic definition of SBZP given here.

Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One (SBO)

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One-or-Preserved (SBOP)

From the introduction of the Armv8 architecture, the description *Should-Be-One-or-Preserved (SBOP)* is superseded by *RES1*.

---- **Note** ----

The Armv7 Large Physical Address Extension modified the definition of SBOP for register bits that are SBOP in some but not all contexts. The behavior of these bits is covered by the *RES1* definition, but not by the generic definition of SBOP given here.

Hardware must ignore writes to the field.

When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

RISC   Reduced Instruction Set Computer.

SBZP   *See* Should-Be-Zero-or-Preserved (SBZP).

SBZ   *See* Should-Be-Zero (SBZ).

Should-Be-Zero (SBZ)

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Simple sequential execution

The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Translation   Defines the process of generating a valid output memory address from an input address. It also defines the behavior when it is not possible to generate a valid output address. Translation can be implemented using an MMU or an MPU.

Translation table

A table held in memory that defines the properties of memory areas of various sizes from 1KB to 1MB.

Translation table walk

The process of doing a full translation table lookup. It is performed automatically by hardware.

UNDEFINED   Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the UNDEFINED encoding was executed at EL0. This applies to:

•   Any encoding that is not allocated to any instruction.

•   Any encoding that is defined as never accessible at the current Exception level.

•   Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

If the generated exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

—— **Note** ——

On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive.

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value can vary from moment to moment, and instruction to instruction, unless it has previously been assigned, other than at reset, to one of the following registers:

- Any of the general-purpose registers.
- Any of the Advanced SIMD and floating-point registers.
- Any of the Scalable Vector Extension registers.
- Any of the PSTATE N, Z, C, or V flags.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE.

UNPREDICTABLE

Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

Validation

In an address translation context, validation refers to translation that is implemented by the MPU in which the input address and output address are always the same.

Virtual address (VA)

An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also Intermediate physical address (IPA), Physical address (PA).