ARM Architecture Reference Manual Supplement
ARMv8, for the ARMv8-R AArch32 architecture profile

Copyright © 2016, 2017 ARM Limited or its affiliates. All rights reserved.

Release Information
The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Aug 2016</td>
<td>A.a</td>
<td>Confidential</td>
<td>LAC release</td>
</tr>
<tr>
<td>31 Mar 2017</td>
<td>A.b</td>
<td>Non-Confidential</td>
<td>EAC release</td>
</tr>
</tbody>
</table>

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

Copyright © 2016, 2017 ARM Limited or its affiliates. All rights reserved.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

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

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

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

Web Address

http://www.arm.com
Contents

ARM Architecture Reference Manual Supplement
ARMv8, for the ARMv8-R AArch32 architecture profile

Preface
Using this book ............................................................................................................ x
Conventions ............................................................................................................... xi
Additional reading ................................................................................................... xii
Feedback .................................................................................................................. xiii

Part A
ARMv8-R Architecture Introduction and Overview

Chapter A1
ARMv8-R Introduction and Overview
A1.1 About the ARMv8 architecture and architecture profiles ......................... A1-18
A1.2 The ARMv8-R AArch32 architecture profile .............................................. A1-19
A1.3 Supported extensions in ARMv8-R AArch32 ............................................ A1-20
A1.4 Changes between ARMv7-R and ARMv8-R AArch32 ............................ A1-21

Part B
Differences in the ARMv8-R Architecture from ARMv8-A

Chapter B1
Differences between the ARMv8-A and ARMv8-R AArch32 Profiles
B1.1 Differences from the ARMv8-A AArch32 application level architecture ........ B1-26
B1.2 Differences from the ARMv8-A AArch32 instruction sets ............................ B1-27
B1.3 Differences from the ARMv8-A AArch32 system level architecture ............. B1-28
Part I  Appendixes

Appendix I1  ARMv8-R AArch32 CONSTRAINED UNPREDICTABLE behaviors
I1.1  Reserved values in System registers and memory attribute settings ............... 11-354
Preface

This preface introduces the ARM® Architecture Reference Manual Supplement ARMv8, for the ARMv8-R AArch32 architecture profile. It contains the following sections:

- Using this book on page x.
- Conventions on page xi.
- Additional reading on page xii.
- Feedback on page xiii.
Using this book

This supplement describes the operation of an ARMv8-R *processing element* (PE) in AArch32 state. It describes the ARMv8-R profile in terms of how it differs from the ARMv8-A profile.

This book is a supplement to the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* (DDI0487), and is intended to be used with it. It is assumed that the reader is familiar with the ARMv8 architecture.

This manual is organized into parts that follow the order and structure in the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*:

**Part A**
Provides an introduction to the ARMv8-R architecture.

**Part B**
Describes the non-debug features in which the ARMv8-R profile differs from the ARMv8-A profile.

**Part C**
Describes the *ARMv8-R Protected Memory System Architecture* (PMSAv8).

**Part D**
Describes the ARMv8-R instruction set.

**Part E**
Describes the ARMv8-R System registers and System instructions.

**Part F**
Describes the debug features in which the ARMv8-R profile differs from the ARMv8-A profile.

**Part G**
Describes the ARMv8-R debug registers.

**Part H**
Describes the ARMv8-R pseudocode.

**Appendixes**
Provide additional information.
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:

<table>
<thead>
<tr>
<th>Style</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>italic</td>
<td>Introduces special terminology, and denotes citations.</td>
</tr>
<tr>
<td>bold</td>
<td>Denotes signal names, and is used for terms in descriptive lists, where appropriate.</td>
</tr>
</tbody>
</table>
| 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 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 http://infocenter.arm.com.                          |
|               | • A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, 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 0x000000FF.

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 the Infocenter http://infocenter.arm.com and http://developer.arm.com, for access to ARM documentation.

ARM publications

- ARM® CoreSight™ Architecture Specification (ARM IHI 0029).
- ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0 (ARM IHI 0069).

Other publications

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

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

• The title.
• The number, DDI0568A.b.
• The page numbers to which your comments apply.
• A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

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

ARMv8-R Architecture Introduction and Overview
Chapter A1
ARMv8-R Introduction and Overview

This chapter introduces the ARMv8 architecture, the architecture profiles it defines, and the ARMv8-R AArch32 profile that this manual defines. It contains the following sections:
• About the ARMv8 architecture and architecture profiles on page A1-18.
• The ARMv8-R AArch32 architecture profile on page A1-19.
• Supported extensions in ARMv8-R AArch32 on page A1-20.
• Changes between ARMv7-R and ARMv8-R AArch32 on page A1-21.
A1 ARMv8-R Introduction and Overview
A1.1 About the ARMv8 architecture and architecture profiles

A1.1 About the ARMv8 architecture and architecture profiles

The ARM architecture that this Architecture Reference Manual describes defines the behavior of an abstract machine, referred to as a processing element (PE). 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. For more information, see the Embedded Trace Macrocell Architecture Specification.

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.

ARMv8 defines three architecture profiles:

A Application profile:

- Supports a Virtual Memory System Architecture (VMSA) based on a Memory Management Unit (MMU).
- Supports the A64, A32, and T32 instruction sets.

R Real-time profile, described in this manual:

- Supports a Protected Memory System Architecture (PMSA) based on a Memory Protection Unit (MPU).
- Supports the A32 and T32 instruction sets.

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.

This Architecture Reference Manual Supplement:

- Describes the ARMv8-R profile in terms of how it differs from the ARMv8-A profile. If a definition is not mentioned in this Supplement, the definition in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile applies.
- Does not describe details of other profiles but assumes knowledge of the ARMv8-A profile.
- Describes differences between the ARMv7-R and ARMv8-R profiles.
A1.2 The ARMv8-R AArch32 architecture profile

The main features of the ARMv8-R AArch32 profile are:

- One Execution state, AArch32.
- The A32 and T32 instruction sets, which are compatible with earlier versions of the ARM architecture.
- The Protected Memory System Architecture (PMSA) that defines memory ordering and memory management with a single 32-bit physical address (PA) space. The ARMv8-R PMSA is not compatible with the ARMv7-R architecture.
- A programmers’ model and its interfaces to System registers that control most PE and memory system features and provide status information.
- Support for Advanced SIMD and floating-point instructions (optional).
- The ARMv8-R virtualization model.
- The ARMv8 Debug architecture that provides software access to debug features.
A1.3 Supported extensions in ARMv8-R AArch32

A1.3.1 Security extensions

ARMv8-R only supports a single Security state, Non-secure.

A1.3.2 ARMv8.x extensions for the ARMv8-A profile

ARMv8-R does not support any of the ARMv8.x extensions.

A1.3.3 Advanced SIMD and floating-point extensions

Both the Advanced SIMD and floating-point instructions are OPTIONAL. When they are implemented, they must conform to the ARMv8-A AArch32 specifications.

The inclusion of double-precision floating-point processing is also OPTIONAL. When only the single-precision encoding format is implemented and Advanced SIMD is not implemented, the floating-point implementation must be a D16 implementation.

For more information, see Advanced SIMD and floating-point support on page B1-26 and Single-precision only floating-point implementations on page D1-54.
A1.4 Changes between ARMv7-R and ARMv8-R AArch32

- The architecture is described in terms of the ARMv8 Exception levels. ARMv8-R implementations support EL0, EL1, and EL2.

- ARMv8-R supports virtualization, which provides:
  - Support for the EL2 Exception level.
  - A second MPU that provides stage 1 memory protection for memory accesses from EL2 and optionally provides stage 2 memory protection for accesses from EL1 and EL0. These protection stages act as address translation regimes in the ARMv8-R AArch32 profile.

- A redefined PMSA, PMSAv8, provides:
  - A new model for defining protection regions, using a pair of 32-bit addresses to define the region.
  - An increase of the minimum protection region size from 32 bytes to 64 bytes.
  - No support for subregions or overlapping regions.

- There is no CPACR.D32DIS control in ARMv8-R floating-point implementations.

- ARMv8-R supports only a trivial Jazelle implementation.

- Divide by zero trapping is not supported in hardware.
A1 ARMv8-R Introduction and Overview
A1.4 Changes between ARMv7-R and ARMv8-R AArch32
Part B

Differences in the ARMv8-R Architecture from ARMv8-A
Chapter B1
Differences between the ARMv8-A and ARMv8-R AArch32 Profiles

This chapter describes the ARMv8-R AArch32 profile. It contains the following sections:
• Differences from the ARMv8-A AArch32 application level architecture on page B1-26.
• Differences from the ARMv8-A AArch32 instruction sets on page B1-27.
• Differences from the ARMv8-A AArch32 system level architecture on page B1-28.
B1 Differences between the ARMv8-A and ARMv8-R AArch32 Profiles
B1.1 Differences from the ARMv8-A AArch32 application level architecture

B1.1 Differences from the ARMv8-A AArch32 application level architecture

B1.1.1 Advanced SIMD and floating-point support

In addition to the Advanced SIMD and floating-point support in ARMv8-A, ARMv8-R AArch32 also supports single-precision floating-point implementations without Advanced SIMD. If an implementation supports only single-precision floating-point operations and does not support Advanced SIMD, then it must implement only 16 double-precision registers, D0-D15, that is, it must be a D16 implementation. See also ARMv8-R Advanced SIMD and floating-point instructions on page D1-53 and Single-precision only floating-point implementations on page D1-54.

Table B1-1 shows the field values that are permitted in ARMv8-R in addition to the values that are permitted in ARMv8-A.

Table B1-1 Advanced SIMD and floating-point ID field values additionally permitted in ARMv8-R

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVFR0.FPDP</td>
<td>0000</td>
<td>No double-precision support.</td>
</tr>
<tr>
<td>MVFR0.SIMDReg</td>
<td>0001</td>
<td>16x 64-bit register SIMD&amp;FP register file.</td>
</tr>
<tr>
<td>MVFR1.FPHP</td>
<td>0001</td>
<td>Support for half-precision to and from single-precision conversion (no double-precision).</td>
</tr>
<tr>
<td>MVFR1.SIMDIHP</td>
<td>0000</td>
<td>No Advanced SIMD</td>
</tr>
<tr>
<td>MVFR1.SIMDSP</td>
<td>0000</td>
<td>No Advanced SIMD</td>
</tr>
<tr>
<td>MVFR1.SIMDInt</td>
<td>0000</td>
<td>No Advanced SIMD</td>
</tr>
<tr>
<td>MVFR1.SIMDLS</td>
<td>0000</td>
<td>No Advanced SIMD</td>
</tr>
</tbody>
</table>

The ARMv8-A definitions of HCPTR.TASE, NSACR, and CPACR apply for all implementations, including D16.

B1.1.2 Differences from the ARMv8-A AArch32 application level programmers’ model

The following are the differences from the ARMv8-A profile as it is described in chapter The AArch32 Application Level Programmers’ Model in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

- ARMv8-R only supports a single Security state.
- ARMv8-R implementations cannot include EL3.
- EL2 is required in ARMv8-R.
- Exception levels in ARMv8-R are based on the ARMv8-A AArch32 PE modes. Hyp mode provides the EL2 functionality.
- ARMv8-R supports the A32 and T32 instruction sets with some modifications, see Part D ARMv8-R Instructions.

B1.1.3 Differences from the ARMv8-A AArch32 application level memory model

ARMv8-R redefines DMB and DSB, and defines a new instruction, DFB.

ARMv8-R relaxes the ordering requirements for DMB and DSB by enforcing ordering only in terms of certain Exception level criteria:

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

All other memory barrier requirements are unchanged. See Memory barriers in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
B1.2 Differences from the ARMv8-A AArch32 instruction sets

See Chapter D1 ARMv8-R Instruction Set.
B1.3 Differences from the ARMv8-A AArch32 system level architecture

B1.3.1 Differences from the ARMv8-A AArch32 system level programmers’ model

Virtualization
ARMv8-R includes a modified form of the ARMv8-A AArch32 virtualization scheme. In common with AArch32 state in ARMv8-A, EL2 in ARMv8-R AArch32 supports virtualization and adds Hyp mode to the implemented PE modes. The relationship between Hyp mode and the other implemented PE modes is the same as for an ARMv8-A AArch32 implementation that does not include EL3. See Memory protection units (MPUs) on page C1-32.

GIC
A Generic Interrupt Controller (GIC) implemented with an ARMv8-R PE must not implement LPI support.

B1.3.2 Differences from the ARMv8-A AArch32 system level memory model

Address space
ARMv8-R uses a 32-bit address space with a flat mapping from the virtual address (VA) used by the PE to the physical address (PA).

Address translation
In ARMv8-R, address translation is the process of flat-mapping a VA to a PA and determining 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 (TCM) to EL1 and EL0 or just EL2 is IMPLEMENTATION DEFINED.

B1.3.3 The ARMv8-R Protected Memory System Architecture, PMSAv8
This Supplement defines PMSAv8. See Chapter C1 ARMv8-R Protected Memory System Architecture.
Part C

ARMv8-R Protected Memory System Architecture
Chapter C1
ARMv8-R Protected Memory System Architecture

This chapter provides a system-level view of the memory system architecture for an ARMv8-R implementation, the Protected Memory System Architecture (PMSAv8-32). It contains the following sections:

- About PMSAv8-32 on page C1-32.
- Protection region attributes and access permissions on page C1-35.
- Default memory maps and Background region checks on page C1-38.
- Memory protection on page C1-40.
- PMSAv8-32 implications for caches on page C1-43.
C1.1 About PMSAv8-32

For general information about the ARM memory model, see chapters The AArch32 Application Level Memory Model and The AArch32 System Level Memory Model in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

An ARM PMSA is based on a Memory Protection Unit (MPU) that provides a memory protection scheme. The PMSA uses a flat mapping between the virtual address (VA) accessed by the PE and the 32-bit physical address (PA) accessed in the memory system. That is, for all accesses, the VA is the same as the PA, see Address translation and translation regimes in PMSAv8-32 on page C1-33.

PMSAv8-32 only supports a unified memory protection scheme. It does not support separate instruction and data regions in the address map. PMSAv8-32 is not backwards-compatible with PMSAv7 or earlier ARM PMSAs.

C1.1.1 Protection regions

An MPU defines protection regions in the 4GB address map. A protection region is a contiguous memory region for which the MPU defines the memory attributes, for example the memory type and the access permissions.

Protection regions:
• Are defined by a pair of a Base Address Register and a Limit Address Register, see Memory protection units (MPUs).
• Have a minimum size of 64 bytes.
• Have a maximum size of the entire address map, 4GB.
• Must not overlap.
• Do not need to be contiguous.

The definition of a protection region specifies the start and the end of the region, the access permissions to the region, and the memory attributes for the region. For more information, see Memory protection on page C1-40.

C1.1.2 Memory protection units (MPUs)

PMSAv8-32 defines two MPUs:

EL1-controlled MPU

Defines the protection regions for accesses from EL1 and from EL0.

EL2-controlled MPU

Defines the protection regions for accesses from EL2.

When the value of HCR.VM is 1 and for accesses from EL1 and from EL0, the EL2-controlled MPU uses these protection regions to modify the access permissions and memory attributes that are assigned by the EL1-controlled MPU.

See Protection region attributes and access permissions on page C1-35. For an address that does not match any defined protection region, PMSAv8-32 defines Background regions and default memory maps. See Default memory maps and Background region checks on page C1-38.

An PMSAv8-32 implementation can provide a virtualization scheme in which a single PE supports multiple guest environments under the control of a single hypervisor that executes at EL2, where executing at EL2 means executing in Hyp mode. Typically, an operating system executing at EL1 programs the EL1-controlled MPU to define the memory map for its own operation and for applications it runs at EL0. A hypervisor executing at EL2 programs the EL2-controlled MPU to define the memory map for its own operation and to modify the access permission and memory attribute assignments that are performed by the EL1&0 stage 1 translation.
C1.1.3 Address translation and translation regimes in PMSAv8-32

In ARMv8-R:

- Address translation describes the process of flat mapping the virtual address (VA) used by the PE to the physical address (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).

**Note**

These definitions provide consistency between the ARMv8-A and ARMv8-R descriptions of the memory model.

PMSAv8-32 defines the following translation regimes:

**EL1&0 translation regime**

This 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-controlled MPU, if it is enabled. This translation is a stage 1 translation.

- When the value of HCR.VM is 1, the access is also translated by the EL2-controlled 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 ADDRESS is in protection region n if:

\[ \text{PRBAR}_n.\text{BASE}:'000000' \leq \text{ADDRESS} \leq \text{PRLAR}_n.\text{LIMIT}:'111111' \]

**EL2 translation regime**

This 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-controlled MPU.

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

\[ \text{HPBAR}_n.\text{BASE}:'000000' \leq \text{ADDRESS} \leq \text{HPRLAR}_n.\text{LIMIT}:'111111' \]

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

- The values that are programmed into the Base Address Register and Limit Address Register pair that define the protection region.

- A Memory Attributes Indirection register that is indexed by those values.

*Figure C1-1 on page C1-34 shows how memory accesses are handled when both MPUs are enabled. For other scenarios, see Effect of enabling one or both MPUs on attribute assignment and fault generation on page C1-40.*
C1 ARMv8-R Protected Memory System Architecture
C1.1 About PMSAv8-32

Figure C1-1 PMSAv8-32 memory access permission and attribute control

- Controlled from EL2
- Controlled from EL1
- Access from EL1 is not allowed when HCR.TGE==1
C1.2 Protection region attributes and access permissions

The protection region attribute fields control the memory type, Cacheability, and Shareability of the region. ARMv8-R uses the same memory types and memory attributes as ARMv8-A. The ARMv8-R identification scheme is based on the scheme that is used by VMSAv8-32 when using the Long-descriptor translation table format. See VMSAv8-32 Long-descriptor format memory region attributes in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

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

- The PRBAR and PRLAR, or HPRBAR and HPRLAR, that define the region.
- The MAIR\(^<m>\).Attr\(^<n>\) or HMAIR\(^<m>\).Attr\(^<n>\) field that is indexed by PRLAR.AttrIndx or HPRLAR.AttrIndx, respectively. In ARMv8-R, the PE always behaves as if the value of TTBCR.EAE were 1 even though the encoding space that TTBCR uses in ARMv8-A is unallocated in ARMv8-R. For information about the behavior when TTBCR.EAE is 1, see the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

See also Assignment model of memory attributes and access permissions.

For the EL1&0 translation regime, when both MPUs are enabled and the value of HCR.VM is 1, the stage 1 memory attribute and access permission assignments are combined with the stage 2 assignments as described in Combining attributes on page C1-36.

C1.2.1 Assignment model of memory attributes and access permissions

**PRBAR.SH, HPRBAR.SH**

Defines the Shareability, for a Normal memory protection region. For any type of Device memory, and for Normal Inner Non-cacheable, Outer Non-cacheable memory, the value of the SH[1:0] field is IGNORED.

**PRBAR.AP[2:1], HPRBAR.AP[2:1]**

Defines the Access permissions. For encoding information, see the register description.

**PRBAR.XN, HPRBAR.XN**

Defines the Execute-never attribute for the region. For encoding information, see the register description.

\(XN = 1\) The region is Execute-never.

PMSAv8-32 makes no distinction between Privileged execute-never (PXN) and Unprivileged execute-never (UXN).

**PRLAR.AttrIndx, HPRLAR.AttrIndx**

Indexes an Attr\(^<n>\) field in MAIR\(^<m>\) or HMAIR\(^<m>\), as follows:

- AttrIndx[2] indicates the MAIR register to be used:
  - AttrIndx[2] == 0 Use MAIR0 or HMAIR0.
  - AttrIndx[2] == 1 Use MAIR1 or HMAIR1.

- AttrIndx[2:0] indicates the required Attr field, Attr\(^<n>\), where \(n = \text{AttrIndx}[2:0]\).

**MAIR\(^<m>\).Attr\(^<n>\), HMAIR\(^<m>\).Attr\(^<n>\)**

Defines:

- The memory type, Normal memory or a type of Device memory.
- For Normal memory:
  - The Inner Cacheability and Outer Cacheability attributes, each of which is one of Non-cacheable, Write-Through Cacheable, or Write-Back Cacheable.
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is Allocate or Do not allocate.
C1.2.2 Combining attributes

For the EL1&0 translation regime, when both MPUs are enabled and the value of HCR.VM is 1, the stage 1 and stage 2 attributes and access permissions are combined as follows:

• If the stage 1 access permissions indicate that an access is not permitted, a stage 1 Permission fault is generated regardless of the stage 2 permissions.

• If the stage 1 access permissions indicate that an access is permitted but the stage 2 access permissions indicate that it is not permitted, a stage 2 Permission fault is generated.

• If an access is permitted by both the stage 1 and the stage 2 access permissions, the memory attributes are combined as shown in Table C1-1, Table C1-2, and Table C1-3 on page C1-37.

Table C1-1 Combining the stage 1 and stage 2 memory type assignments

<table>
<thead>
<tr>
<th>Rule</th>
<th>If the memory type assigned by either stage is:</th>
<th>The resultant memory type is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device has precedence over Normal</td>
<td>Any Device memory type</td>
<td>A Device memory type</td>
</tr>
<tr>
<td>Non-Gathering has precedence over Gathering</td>
<td>A Device-nGxx memory type</td>
<td>A Device-nGxx memory type</td>
</tr>
<tr>
<td>Non-Reordering has precedence over Reordering</td>
<td>A Device-nGnRx memory type</td>
<td>A Device-nGnRx memory type</td>
</tr>
<tr>
<td>No Early Write Acknowledgement has precedence over Early Write Acknowledgement</td>
<td>The Device-nGnRnE memory type</td>
<td>The Device-nGnRnE memory type</td>
</tr>
</tbody>
</table>

Regardless of any Shareability attribute that results from the combinations that are described in Table C1-3 on page C1-37:

• Any location for which the resultant memory type is any type of Device memory is always treated as Outer Shareable.

• Any location for which the resultant memory type is Normal Inner Non-cacheable, Outer Non-cacheable is always treated as Outer Shareable.

Table C1-2 shows the assignments at each stage of translation are combined to create the resultant Cacheability attribute. These rules apply independently for the Inner Cacheability and Outer Cacheability attributes.

Table C1-2 Combining Cacheability assignments for Normal memory

<table>
<thead>
<tr>
<th>Stage 1 assignment</th>
<th>Stage 2 assignment</th>
<th>Resultant Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-cacheable</td>
<td>Any</td>
<td>Non-cacheable</td>
</tr>
<tr>
<td>Any</td>
<td>Non-cacheable</td>
<td>Non-cacheable</td>
</tr>
<tr>
<td>Write-Through Cacheable</td>
<td>Write-Through or Write-Back Cacheable</td>
<td>Write-Through Cacheable</td>
</tr>
<tr>
<td>Write-Through or Write-Back Cacheable</td>
<td>Write-Through Cacheable</td>
<td>Write-Through Cacheable</td>
</tr>
<tr>
<td>Write-Back Cacheable</td>
<td>Write-Back Cacheable</td>
<td>Write-Back Cacheable</td>
</tr>
</tbody>
</table>

A protection region is treated as Outer Shareable, regardless of any Shareability assignments at either stage of translation if:

• The resultant memory type attribute, as described in Table C1-1, is any type of Device memory.

• The resultant memory type attribute, as described in Table C1-1, is Normal memory, and the resultant Cacheability, as described in Table C1-2, is Inner Non-cacheable, Outer Non-cacheable.
For a protection region with a resultant memory type attribute of Normal that is not Inner Non-cacheable, Outer Non-cacheable, Table C1-3 shows how the Shareability attribute is obtained from the assignment at each stage of translation.

Table C1-3 Combining Shareability assignments for Normal not Inner Non-cacheable, Outer Non-cacheable memory

<table>
<thead>
<tr>
<th>Stage 1 assignment</th>
<th>Stage 2 assignment</th>
<th>Resultant Shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer Shareable</td>
<td>Any</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Non-shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Non-shareable</td>
<td>Non-shareable</td>
</tr>
</tbody>
</table>
C1.3 Default memory maps and Background region checks

Each PMSA\textsuperscript{v8-32} MPU has an associated default memory map which is used when the MPU is not enabled.

When the MPU is enabled and Background region checking is enabled, privileged accesses that do not hit defined protection regions undergo a second check.

- For the EL1 MPU, Background region checking is enabled for privileged access when the value of SCTL\textsubscript{R}.BR is 1.
- For the EL2 MPU, Background region checking is enabled for accesses from EL2 when the value of HSCTL\textsubscript{R}.BR is 1.

For details, see Effect of enabling one or both MPUs on attribute assignment and fault generation on page C1-40.

C1.3.1 EL1 MPU default memory map

Table C1-4 and Table C1-5 describe the default memory map defined for the EL1 MPU. When the value of HCR.DC is 0, Background region checking by the EL1 MPU also uses this memory map.

Table C1-4 Instruction accesses and EL1 Background region checks

<table>
<thead>
<tr>
<th>Address range</th>
<th>SCTL\textsubscript{R}.I==0</th>
<th>SCTL\textsubscript{R}.I==1</th>
<th>XN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 – 0xFFFFFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Non-shareable, Write-Through Cacheable</td>
<td>Execution permitted</td>
</tr>
<tr>
<td>0x80000000 – 0xFFFFFFFF</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Execute-never</td>
</tr>
</tbody>
</table>

Table C1-5 Data accesses and EL1 Background region checks

<table>
<thead>
<tr>
<th>Address range</th>
<th>SCTL\textsubscript{R}.C==0</th>
<th>SCTL\textsubscript{R}.C==1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 – 0x3FFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Non-shareable, Write-Back, Write-Allocate Cacheable</td>
</tr>
<tr>
<td>0x40000000 – 0x5FFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Non-shareable, Write-Through Cacheable</td>
</tr>
<tr>
<td>0x60000000 – 0x7FFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Shareable, Non-cacheable</td>
</tr>
<tr>
<td>0x80000000 – 0xBFFFFF</td>
<td>Device-nGnRE</td>
<td>Device-nGnRE</td>
</tr>
<tr>
<td>0xC0000000 – 0xFFFFFFFF</td>
<td>Device-nGnRnE</td>
<td>Device-nGnRnE</td>
</tr>
</tbody>
</table>

When the value of HCR.DC is 1, the Background region and the Default memory map produce accesses with the following attributes:

- Normal memory.
- Non-shareable.
- Inner Write-Back Read-Allocate, Write-Allocate Cacheable.
- Outer Write-Back Read-Allocate, Write-Allocate Cacheable.
- Execution permitted.
C1.3.2  EL2 MPU default memory map

Table C1-6 and Table C1-7 describe the default memory map defined for the EL2 MPU. Background region checking by the EL2 MPU also uses this memory map.

Table C1-6 Instruction accesses and EL2 Background region checks

<table>
<thead>
<tr>
<th>Address range</th>
<th>HSCTL.R.I==0</th>
<th>HSCTL.R.I==1</th>
<th>XN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 – 0x7FFFFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Non-shareable, Write-Through Cacheable</td>
<td>Execution permitted</td>
</tr>
<tr>
<td>0x80000000 – 0xFFFFFFFF</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Execute-never</td>
</tr>
</tbody>
</table>

Table C1-7 Data accesses and EL2 Background region checks

<table>
<thead>
<tr>
<th>Address range</th>
<th>HSCTL.R.C==1</th>
<th>HSCTL.R.C==1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 – 0x3FFFFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Non-shareable, Write-Back, Write-Allocate Cacheable</td>
</tr>
<tr>
<td>0x40000000 – 0x5FFFFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Non-shareable, Write-Through Cacheable</td>
</tr>
<tr>
<td>0x60000000 – 0x7FFFFFFF</td>
<td>Normal, Shareable, Non-cacheable</td>
<td>Normal, Shareable, Non-cacheable</td>
</tr>
<tr>
<td>0x80000000 – 0xBFFFFFFF</td>
<td>Device-nGnRE</td>
<td>Device-nGnRE</td>
</tr>
<tr>
<td>0xC0000000 – 0xFFFFFFFF</td>
<td>Device-nGnRnE</td>
<td>Device-nGnRnE</td>
</tr>
</tbody>
</table>
C1.4 Memory protection

An MPU checks whether the address used by a memory access matches a defined protection region. It uses the properties that are defined for that region or for the Background region to determine whether the access is permitted and if it is permitted, how it must behave.

- The EL1 MPU is enabled when the value of SCTLR.M is 1.
- The EL2 MPU is enabled when the value of HSCTLR.M is 1.

Each MPU provides two mechanisms for defining its own protection regions by:

- Using indirect accesses, where:
  - PRSELR/HPRSELR is programmed to specify the required protection region.
  - PRBAR/HPRBAR and PRLAR/HPRLAR are programmed to specify the address range and attributes for that region.
- Directly specifying the address range and attributes for the first 32 regions using PRBAR<n>/HPRBAR<n> and PRLAR<n>/HPRLAR<n>.

For the EL2 protection regions, HPRENR provides direct access to the region enable for regions 0-31.

C1.4.1 Effect of enabling one or both MPUs on attribute assignment and fault generation

For accesses that fault:

- MPU-generated faults follow the PMSA\textsuperscript{v8-32} Long-descriptor format and are reported in the same way as VMSA\textsuperscript{v8-32} faults.
- All MPU faults are treated as level 0 faults. For translation operations, PMSA contexts return the 64-bit PAR format. The PAR format discards the least significant bits but the translation query uses the entire input address. For more information, see MMU faults in AArch32 state in the ARM\textsuperscript{TM} Architecture Reference Manual, ARM\textsuperscript{v8}, for ARM\textsuperscript{v8}-A architecture profile. For fault status codes, see DFSR, IFSR, and HSR.
- If the value of EDSCR.HDD is 1, stage 2 faults in Debug state are reported as Debug events at EL1. See Behavior in Debug state on page F1-189.

For an access that does not fault:

When both the EL1-controlled MPU and the EL2-controlled MPU are enabled:

- For an EL2 access (an access from Hyp mode), the EL2 MPU configuration settings determine the access permissions and memory attributes.
- For EL1 and EL0 accesses, the EL1 MPU configuration settings determine the stage 1 access permissions and memory attributes. If the Effective value of HCR.VM is 1, those access permissions and memory attributes are then modified by the EL2-controlled MPU, as described in Combining attributes on page C1-36.

When the EL1-controlled MPU is disabled and the EL2-controlled MPU is enabled:

- For an EL2 access (an access from Hyp mode), the EL2 MPU configuration settings determine the access permissions and memory attributes.
- For an EL1 or EL0 access, the EL1 Background region settings determine the stage 1 access permissions and memory attributes. If the Effective value of HCR.VM is 1, those access permissions and memory attributes are then modified by the EL2-controlled MPU, as described in Combining attributes on page C1-36.

When the EL1-controlled MPU is enabled but the EL2-controlled MPU is disabled:

- For an EL2 access (an access from Hyp mode), the EL2 Background region settings determine the access permissions and memory attributes.
- For EL1 and EL0 accesses, the EL1 MPU configuration settings determine the access permissions and memory attributes. If the Effective value of HCR.VM is 1, those access permissions and memory attributes are then combined with the EL2 Background region settings, as described in Combining attributes on page C1-36.
When both the EL1 MPU-controlled and the EL2-controlled MPU are disabled:

- For an EL2 access (an access from Hyp mode), the EL2 Background region settings determine the stage 1 access permissions and memory attributes.
- For an EL1 or EL0 access, the EL1 Background region settings determine the access permissions and memory attributes. If the Effective value of HCR.VM is 1, those access permissions and memory attributes are then modified by the EL2 Background region settings, as described in Combining attributes on page C1-36.

![Figure C1-2 Stage 1 translation for EL1 and EL0 accesses](image-url)
Figure C1-3 Stage 2 translation for EL1 and EL0, stage 1 translation for EL2 accesses

Combining is trivial if the EL1 MPU check resulted in attributes from the EL1 Background region because the two Background regions are identical when both of them are enabled.
C1.5 PMSAv8-32 implications for caches

Enabling or reconfiguring the MPU, or reprogramming any protection regions, can result in new and different memory attributes for a previously accessed or speculatively accessed address. In this situation, the rules for Mismatched memory attributes apply. See the Mismatched memory attributes in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

C1.5.1 Cache line length

A PMSAv8-32 MPU permits the definition of protection regions that might be smaller than a cache line in the implementation. Therefore, the following rules apply:

- If the MPU is configured such 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. See Memory region attributes in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

- Marking any part of a cache line as Write-Back permits the entire line to be treated as Write-Back.
Chapter D1
ARMv8-R Instruction Set

This chapter describes the T32 and A32 instruction sets of the ARMv8-R profile. It contains the following sections:

- ARMv8-R base instructions on page D1-48.
- ARMv8-R Advanced SIMD and floating-point instructions on page D1-53.
- Single-precision only floating-point implementations on page D1-54.
- Instruction encodings on page D1-56.
D1.1 ARMv8-R base instructions

Table D1-1 summarizes the instructions that are new in the ARMv8-R AArch32 profile or redefined from ARMv8-A. For instructions that are unchanged with respect to ARMv8-A, see the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Status with respect to ARMv8-A</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ADR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>AESD, AESE, AESINC, AESMC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ASR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>BFC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>BFI</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>BKPT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>BL, BLX, BX, BXJ</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CBNZ</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CBZ</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CDP</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CDP2</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CLREX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CLZ</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CMN</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CPS, CPSID, CPSIE</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>CRC32, CRC32C</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>DBG</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>DCP51, DCP52</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>DCP53</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>DFB on page D2-66</td>
<td>New</td>
<td></td>
</tr>
<tr>
<td>DMB on page D2-59</td>
<td>Redefined</td>
<td>Differences from the ARMv8-A AArch32 application level architecture on page B1-26</td>
</tr>
<tr>
<td>DSB on page D2-62</td>
<td>Redefined</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Status with respect to ARMv8-A</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>EOR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ERET</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>HLT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>HVC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>TSB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>IT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LDA, LDA8, LDAEX, LDAEXB, LDAEXD, LDAEXH, LDAB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LDC, LDC2</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LDM, LDMDA, LDMDA, LDMDA, LDMDA, LDMDA</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LDR, LDRB, LDRBT, LDRD</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LDREX, LDREXB, LDREXD, LDREXH, LDRH, LDRHT, LDRSB, LDRSBT, LDRSH, LDRSHT, LDRT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LSL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MCR, MCR2, MCR, MCR2</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MLA</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MLS</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MOV2</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MRC, MRC2, MRRC, MRRC2</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MRS</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MSR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>MVN</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ORN</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ORR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>PKHBT, PKHTB</td>
<td>Unchanged</td>
<td></td>
</tr>
</tbody>
</table>
### Table D1-1 ARMv8-R base instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Status with respect to ARMv8-A</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLD</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>PLI</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>QADD, QADD16, QADD8, QASX, QADD0</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>QSUB, QSUB16, QSUB8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>RBIT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>REV, REV16, REVSH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>RFE, RFEDA, RFEA, RFEIB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>RRX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>RSB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>RSC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SADD16, SADD8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SASX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SBC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SBFX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SDIV</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SEL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SETEND</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SEV</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SEVL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SHA1C, SHA1M, SHA1P, SHA1H, SHA1SU0, SHA1SU1</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SHA256H, SHA256H2, SHA256SU0, SHA256SU1</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SHAADD16, SHAADD8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SHASX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SHSAX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Status with respect to ARMv8-A</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>SHSUB16, SHSUB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SMC</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>SMLABB, SMLABT, SMLATB, SMLATT, SMLAD, SMLADX, SMLAL, SMLALS, SMLALB, SMLALBT, SMLALTB, SMLALTT, SMLALD, SMLALDX, SMLAWB, SMLAWT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMILA, SMMLAR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SMMLS, SMMLSR</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SMMUL, SMMULR, SMUAD, SMUADX, SMUULB, SMULBT, SMULTB, SMULTT, SMULL, SMULLS, SMULWB, SMULWT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SMUSD, SMUSDX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SRS, SRSDA, SRSDB, SRS1A, SRS1B</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SSAT, SSAT16</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SSAX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SSUB16, SSUBB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>STC, STC2</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>STL, STLB, STLH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>STRL, STRLB, STRLH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>STM, STMIA, STMEA, STMIA, STMED, STMDB, STMF, STMIB, STMFA</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>STR, STRB, STRBT, STRD, STRH, STRHT, STRT</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>STRLEX, STRLEXB, STRLEXH, STRLEXD</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SVC</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SXTAB, SXTAB16, SXTAH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>SXTB, SXTB16, SXTH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Status with respect to ARMv8-A</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>TBB, TBH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>TEQ</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UADD16, UADD8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UASX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UBFX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UDF</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UDIV</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UHADD16, UHADD8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UHASX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UHSAX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UHSUB16, UHSUB8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UMAAL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UMULL</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UQADD16, UQADD8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UQASX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UQSAK</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UQSUB16, UQSUB8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>USAD8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>USADA8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>USAT, USAT16</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>USAX</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>USUB16, USUB8</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UXTAB, UXTAB16</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UXTAH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UXTB, UXTB16</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>UXTH</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>WFE</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>WFI</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>YIELD</td>
<td>Unchanged</td>
<td></td>
</tr>
</tbody>
</table>
D1.2 ARMv8-R Advanced SIMD and floating-point instructions

The ARMv8-R AArch32 profile supports Advanced SIMD and floating-point operations. For the instruction descriptions, see the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

ARMv8-R only supports single-precision floating-point operations. See also Advanced SIMD and floating-point support on page B1-26 and Single-precision only floating-point implementations on page D1-54.
D1.3 Single-precision only floating-point implementations

ARMv8-R permits a floating-point implementation that supports only single-precision floating-point instructions. Such an implementation must be a D16 implementation, that is, it must support only floating-point registers D0-D15. Instruction encodings that would access floating-point registers beyond D15 are UNDEFINED.

Table D1-2 shows the A32/T32 instruction encodings that are implemented in a single-precision only floating-point implementation.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>VABS</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VADD (floating-point)</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VCMP</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VCMP</td>
<td>A2/T2</td>
<td></td>
</tr>
<tr>
<td>VCVT (integer to floating-point, floating-point), VCVTR</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VCVTA (floating-point), VCVTN (floating-point), VCVTP (floating-point), VCVTM (floating-point)</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VCVTB, VCVTT</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VDIV</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VFMA, VFMS</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VFNMA, VFNMS</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VLDM, VLDMDB, VLDMIA</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>A2/T2</td>
<td></td>
</tr>
<tr>
<td>VLDR</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>A2/T2</td>
<td></td>
</tr>
<tr>
<td>VMAXNM, VMINNM</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VMLA (floating-point), VMLS (floating-point)</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VM0V (immediate)</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VM0V (register, SIMD)</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VM0V (general-purpose register to scalar)</td>
<td>A1/T1</td>
<td>opc1:opc2 = ‘0x00’</td>
</tr>
<tr>
<td>VM0V (scalar to general-purpose register)</td>
<td>A1/T1</td>
<td>U:opc1:opc2 = ‘0x00’</td>
</tr>
<tr>
<td>VM0V (between general-purpose register and single-precision register)</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VM0V (between two general-purpose registers and two single-precision registers)</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VM0V (between two general-purpose registers and a doubleword floating-point register)</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VMRS</td>
<td>A1/T1</td>
<td></td>
</tr>
<tr>
<td>VMSR</td>
<td>A1/T1</td>
<td></td>
</tr>
</tbody>
</table>
### Table D1-2 ARMv8-R instructions in an ARMv8-R single-precision only implementation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMUL (floating-point)</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VNEG</td>
<td>A2/T2</td>
<td>sz=0</td>
</tr>
<tr>
<td>VNMLA, VNMLS, VMUL</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VPOP</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VPUSH</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VRINTA (floating-point), VRINTN (floating-point), VRINTP (floating-point), VRINTM (floating-point)</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VRINTX (floating-point)</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VRINTZ (floating-point), VRINTR</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VSELEQ, VSELG, VSELGt, VSELVs</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VSQRT</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
<tr>
<td>VSTM, VSTMDB, VSTMIA</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VSTR</td>
<td>A1/T1</td>
<td>all</td>
</tr>
<tr>
<td>VSUB (floating-point)</td>
<td>A1/T1</td>
<td>sz=0</td>
</tr>
</tbody>
</table>
D1.4 Instruction encodings

This section contains the encoding for the instructions that are new in ARMv8-R. For all other encoding information, see ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

D1.4.1 Miscellaneous system

This section describes the encoding of the Miscellaneous system instruction class. This section is decoded from Section Branches and miscellaneous control in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

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

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0010</td>
<td>CLREX</td>
</tr>
<tr>
<td>0011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0100</td>
<td>DSB and DFB</td>
</tr>
<tr>
<td>0101</td>
<td>DMB</td>
</tr>
<tr>
<td>0110</td>
<td>ISB</td>
</tr>
<tr>
<td>0111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1xxx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Chapter D2
Description of Redefined and New Instructions

This chapter contains the description of the instructions that are redefined or new in ARMv8-R from ARMv8-A. It contains the following sections:

- Redefined instructions on page D2-58
- New instruction on page D2-65
D2.1 Redefined instructions

This section contains the description of the instructions that are redefined in ARMv8-R from ARMv8-A.
D2.1 DMB

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

The ordering requirements of DMB are:

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

These ordering requirements are a relaxation from the ARMv8-A behavior of DMB.

A1

\[
\begin{array}{cccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & (1) & (1) & (1) & (1) & (1) & (1) & (1) & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & option
\end{array}
\]

A1 variant

\[ \text{DMB}\{<c>\}{<q>} {<option>} \]

**Decode for this encoding**

// No additional decoding required

T1

\[
\begin{array}{cccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & \hline
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & (1) & (1) & (1) & (1) & 1 & 0 & (0) & 0 & (1) & (1) & (1) & 0 & 1 & 0 & 1 & option
\end{array}
\]

T1 variant

\[ \text{DMB}\{<c>\}{<q>} {<option>} \]

**Decode for this encoding**

// No additional decoding required

**Assembler symbols**

\(<c>\) For encoding A1: see "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Must be AL or omitted.

For encoding T1: see "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

\(<q>\) See "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

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

- SY Full system is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Can be omitted. This option is referred to as the full system DMB. Encoded as option = 0b1111.
Full system is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. SYST is a synonym for ST. Encoded as option = 0b1110.

Full system is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1101.

Inner Shareable is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1011.

Inner Shareable is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1010.

Inner Shareable is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1001.

Non-shareable is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0111.

Non-shareable is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0110.

Non-shareable is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0011.

Outer Shareable is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0010.

Outer Shareable is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0001.

Outer Shareable is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0000.

All other encodings of option are reserved. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

--- Note

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

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

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
            domain = MBReqDomain_OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
            domain = MBReqDomain_InnerShareable;
    DataMemoryBarrier(domain, types);
D2.1.2 DSB

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

The ordering requirements of DSB are:

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

These ordering requirements are a relaxation from the ARMv8-A behavior of DSB.

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

A1

\[
\begin{array}{|cccccccccccccccccc|}
\hline
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & option \\
\hline
\end{array}
\]

A1 variant

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

Decode for this encoding

// No additional decoding required

T1

\[
\begin{array}{|cccccccccccccccccc|}
\hline
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & \\
\hline
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & option \\
\hline
\end{array}
\]

T1 variant

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

Decode for this encoding

// No additional decoding required

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFB</td>
<td>option == '1100'</td>
</tr>
</tbody>
</table>

Assembler symbols

<\c> For encoding A1: see "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Must be AL or omitted.

For encoding T1: see "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

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

**SY** Full system is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Can be omitted. This option is referred to as the full system DSB. Encoded as option = 0b1111.

**ST** Full system is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. SYST is a synonym for ST. Encoded as option = 0b1110.

**LD** Full system is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1101.

**ISH** Inner Shareable is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1011.

**ISHST** Inner Shareable is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1010.

**ISHLD** Inner Shareable is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b1001.

**NSH** Non-shareable is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0111.

**NSHST** Non-shareable is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0110.

**NSHLD** Non-shareable is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0101.

**OSH** Outer Shareable is the required Shareability domain, reads and writes are the required access types in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0011.

**OSHST** Outer Shareable is the required Shareability domain, writes are the required access type in both "Group A" and "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0010.

**OSHLD** Outer Shareable is the required Shareability domain, reads are the required access type in "Group A", and reads and writes are the required access types in "Group B". For more information see "Data Memory Barrier (DMB)" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Encoded as option = 0b0001.

For option = 0b1100, see DFB. All other encodings of option are reserved. It is IMPLEMENTATION DEFINED whether options other than SY and DFB are implemented. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.
--- Note ---

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

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

---

**Operation for all encodings**

if ConditionPassed() then

  EncodingSpecificOperations();

  case option of

  case option of

  when '0001' domain = MBReqDomain_OuterShareable; types = MBReqTypes_Reads;
  when '0010' domain = MBReqDomain_OuterShareable; types = MBReqTypes_Writes;
  when '0011' domain = MBReqDomain_OuterShareable; types = MBReqTypes_All;
  when '0101' domain = MBReqDomain_Nonshareable; types = MBReqTypes_Reads;
  when '0110' domain = MBReqDomain_Nonshareable; types = MBReqTypes_Writes;
  when '0111' domain = MBReqDomain_Nonshareable; types = MBReqTypes_All;
  when '1001' domain = MBReqDomain_InnerShareable; types = MBReqTypes_Reads;
  when '1010' domain = MBReqDomain_InnerShareable; types = MBReqTypes_Writes;
  when '1011' domain = MBReqDomain_InnerShareable; types = MBReqTypes_All;
  when '1101' domain = MBReqDomain_FullSystem; types = MBReqTypes_Reads;
  when '1110' domain = MBReqDomain_FullSystem; types = MBReqTypes_Writes;
  when '1111' domain = MBReqDomain_FullSystem; types = MBReqTypes_All;
  when '1100' domain = MBReqDomain_FullSystem; types = MBReqTypes_All;

  if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
    if HCR.BSU == '11' then
      domain = MBReqDomain_FullSystem;
    if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
      domain = MBReqDomain_OuterShareable;
    if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
      domain = MBReqDomain_InnerShareable;
    if option == '1100' and PSTATE.EL == EL2 then
      DataFullBarrier();
    else
      DataSynchronizationBarrier(domain, types);

D2.2  New instruction

This section contains the description of the instruction that is new in ARMv8-R from ARMv8-A.
D2.2.1 DFB

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.

A1

ARMv8.0-R

\[
\begin{array}{ccccccccccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{ccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0
\end{array}
\]

A1 variant

DFB\{<c>\}\{<q>\}

is equivalent to

DSB\{<c>\}\{<q>\} #12

and is always the preferred disassembly.

T1

ARMv8.0-R

\[
\begin{array}{ccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}
\]

T1 variant

DFB\{<c>\}\{<q>\}

is equivalent to

DSB\{<c>\}\{<q>\} #12

and is always the preferred disassembly.

Assembler symbols

\(<c>\)

For encoding A1: see "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Must be AL or omitted.

For encoding T1: see "Standard assembler syntax fields" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

\(<q>\)

Operation for all encodings

The description of DSB gives the operational pseudocode for this instruction.
D2 Description of Redefined and New Instructions
D2.2 New instruction
Part E

ARMv8-R System Registers and System Instructions
Chapter E1
ARMv8-R System Registers and System Instructions

This chapter contains the list of the System registers and System instructions in ARMv8-R. It contains the following sections:

• ARMv8-R System register list on page E1-72.
• ARMv8-R System instructions on page E1-78.
## E1.1 ARMv8-R System register list

Table E1-1 summarizes the System registers in the ARMv8-R AArch32 profile. It specifies whether the register is unchanged, redefined, new, or not used when the ARMv8-R System registers are compared to the registers that are supported in ARMv8-A. For information on the System registers that are unchanged with respect to ARMv8-A, see the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. Registers that are not listed in Table E1-1 are UNALLOCATED in v8-R.

The following terms describe register access at each Exception level that ARMv8-R supports:

- **RW** Read/write.
- **RO** Read-only.
- **-** Not accessible.
- **RAZ/WI** Read-As-Zero, Writes Ignored.
- **Cfg** Configurable access.
- **WO** Write-only.

### Table E1-1 ARMv8-R System registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Status with respect to ARMv8-A</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTLR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>ACTLR2</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>ADFSR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>AIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>AIFSR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>AMAIR0</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>AMAIR1</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>CCSIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>CLIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>CNTFRQ</td>
<td>Unchanged</td>
<td>RW</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>CNTHCTL</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CNTHP_CTL</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CNTHP_CVAL</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CNTHP_TVAL</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CNTKCTL</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>CNTPCTL</td>
<td>Unchanged</td>
<td>RO</td>
<td>Cfg</td>
<td>Cfg</td>
</tr>
<tr>
<td>CNTP_CTL</td>
<td>Unchanged</td>
<td>RW</td>
<td>Cfg</td>
<td>Cfg</td>
</tr>
<tr>
<td>CNTPCVAL</td>
<td>Unchanged</td>
<td>RW</td>
<td>Cfg</td>
<td>Cfg</td>
</tr>
<tr>
<td>CNTPTVAL</td>
<td>Unchanged</td>
<td>RW</td>
<td>Cfg</td>
<td>Cfg</td>
</tr>
<tr>
<td>CNTVCVT</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>CNTVOFF</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>Cfg</td>
</tr>
<tr>
<td>CNTVCCTL</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>Cfg</td>
</tr>
<tr>
<td>Register</td>
<td>Status with respect to ARMv8-A</td>
<td>EL2</td>
<td>EL1</td>
<td>EL0</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>CNTV_TVAL</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>Cfg</td>
</tr>
<tr>
<td>CONTEXTIDR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>CPACR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>CSSELR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>CTR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>DBGAUTHSTATUS</td>
<td>Redefined</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGBVR&lt;n&gt;</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGBXVR&lt;n&gt;</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGCLAIMCLR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGCLAIMSET</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGDCCINT</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGDEVID</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>DBGDEVID1</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>DBGDEVID2</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>DBGIDDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>DBGDRAR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>DBGDSAR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>DBGDSCRext</td>
<td>Redefined</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGDSCRint</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>DBGDTRRXext</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGDTRRXint</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>DBGDTXext</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGDTXint</td>
<td>Unchanged</td>
<td>WO</td>
<td>WO</td>
<td>WO</td>
</tr>
<tr>
<td>DBGOSDLR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGOSECظر</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGOSLAR</td>
<td>Unchanged</td>
<td>WO</td>
<td>WO</td>
<td>-</td>
</tr>
<tr>
<td>DBGOSLSR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>DBGPRCR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGVCR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DBGWFAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table E1-1 ARMv8-R System registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Status with respect to ARMv8-A</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGWVR&lt;n&gt;</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DFAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DFSR</td>
<td>Redefined</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>DLR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
</tr>
<tr>
<td>DSPSR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
</tr>
<tr>
<td>FCSEIDR</td>
<td>Unchanged</td>
<td>RAZ/WI</td>
<td>RAZ/WI</td>
<td>-</td>
</tr>
<tr>
<td>FPEXC</td>
<td>Unchanged</td>
<td>Cfg</td>
<td>Cfg</td>
<td>-</td>
</tr>
<tr>
<td>FPSCR</td>
<td>Redefined (access only)</td>
<td>Cfg</td>
<td>Cfg</td>
<td>Cfg</td>
</tr>
<tr>
<td>FPSID</td>
<td>Unchanged</td>
<td>Cfg</td>
<td>Cfg</td>
<td>-</td>
</tr>
<tr>
<td>HACR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HACTLR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HACTLR2</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HADFSR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HAIFSR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HAMAIR0</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HAMAIR1</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HCPTR</td>
<td>Redefined</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HCR</td>
<td>Redefined</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HCR2</td>
<td>Redefined</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HDCR</td>
<td>Redefined</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HDFAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HIFAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HMAIR0</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HMAIR1</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HMPUIR</td>
<td>New</td>
<td>RO</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPRBAR</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPRBAR&lt;n&gt;</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPRENR</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPRLAR</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPRLAR&lt;n&gt;</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPRSELR</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HPFAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table E1-1 ARMv8-R System registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Status with respect to ARMv8-A</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>HRMR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HSCTLR</td>
<td>Redefined</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HSR</td>
<td>Redefined</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HSTR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HTPIDR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HVBAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ID_AFR0</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_DFR0</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_ISAR0</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_ISAR1</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_ISAR2</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_ISAR3</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_ISAR4</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_ISAR5</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_MMFR0</td>
<td>Redefined</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_MMFR1</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_MMFR2</td>
<td>Redefined</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_MMFR3</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_MMFR4</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_PFR0</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>ID_PFR1</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
</tbody>
</table>

Complying with the general differences between the architecture profiles, the following field value settings apply in ARMv8-R AArch32:

- **Virtualization**: 0b0001
- **Virt_frac**: 0b0000
- **Security**: 0b0000
- **Sec_frac**: 0b0001

<table>
<thead>
<tr>
<th>Register</th>
<th>Status with respect to ARMv8-A</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>IFSR</td>
<td>Redefined</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>ISR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>JIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>JMCR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>JOSCR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>MAIR0</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>Status with respect to ARMv8-A</td>
<td>EL2</td>
<td>EL1</td>
<td>EL0</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>MAIR1</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>MIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>MPIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>MPUIR</td>
<td>New</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>MVBAR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>MVFR0</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>MVFR1</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>MVFR2</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>NMRR</td>
<td>Unused</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>NSACR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PAR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCCFILTR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCCNTR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCCNTR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCEID1</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCNTENCLR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCNTENCLR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMINTNSET</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMCR</td>
<td>Redefined</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMEVCTR&lt;n&gt;</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMINTNCLR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMINTNSET</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMOVSR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMOVSSSET</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMSWINC</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMUSERENR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMXEVCTR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PMXEVTYPER</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>Cfg</td>
</tr>
<tr>
<td>PRBAR</td>
<td>New</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>PRBAR&lt;n&gt;</td>
<td>New</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>PRLAR</td>
<td>New</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
</tbody>
</table>

Table E1-1 ARMv8-R System registers (continued)
### Table E1-1 ARMv8-R System registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Status with respect to ARMv8-A</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRLAR&lt;n&gt;</td>
<td>New</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>PRSELR</td>
<td>New</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>PRRR</td>
<td>Unused</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REVIDR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>RMR</td>
<td>Unused</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RVBAR</td>
<td>Unchanged</td>
<td>RO</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>SCTLR</td>
<td>Redefined</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>SPSR</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCMTR</td>
<td>Unchanged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPIDRPRW</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>TPIDRULO</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>TPIDRURW</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
</tr>
<tr>
<td>VBAR</td>
<td>Unchanged</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
</tr>
<tr>
<td>VMPIDR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VPIDR</td>
<td>Unchanged</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VSCTLR</td>
<td>New</td>
<td>RW</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
E1.2 ARMv8-R System instructions

System instructions that are in use in ARMv8-R AArch32 are unchanged from the ARMv8-A profile. ID_MMFR2.UniTLB == 0b0000 ensures that the TLB-related system instructions are UNALLOCATED.

For information on the System instructions, see the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.*
Chapter E2
Description of the Redefined or New System Registers

This chapter contains the description of the System registers that are new or redefined in ARMv8-R from ARMv8-A. It contains the following sections:

- Redefined System registers on page E2-80.
- New System registers on page E2-154.
E2.1 Redefined System registers

This section contains the description of the System registers that are redefined in ARMv8-R from ARMv8-A.
E2.1.1 DBGAUTHSTATUS, Debug Authentication Status register

The DBGAUTHSTATUS characteristics are:

Purpose
Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configurations
System register DBGAUTHSTATUS is architecturally mapped to External register DBGAUTHSTATUS_EL1.
This register is required in all implementations.

Attributes
DBGAUTHSTATUS is a 32-bit register.

Field descriptions
The DBGAUTHSTATUS bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>HNID</td>
<td>HID</td>
<td>SNID</td>
<td>SID</td>
<td>NSID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:12]
Reserved, RES0.

HNID, bits [11:10]
Hyp non-invasive debug. Possible values of this field are:
- 00 Separate Hyp enable not implemented, or EL2 not implemented.
- 10 Implemented and disabled. ExternalHypNoninvasiveDebugEnabled() == FALSE.
- 11 Implemented and enabled. ExternalHypNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.
For ARMv8-R:
- If EL2 is implemented, bit [11] is RAO.
- If EL2 is not implemented, bits [11:10] are RAZ.

HID, bits [9:8]
Hyp invasive debug. Possible values of this field are:
- 00 Separate Hyp enable not implemented, or EL2 not implemented.
- 10 Implemented and disabled. ExternalHypInvasiveDebugEnabled() == FALSE.
- 11 Implemented and enabled. ExternalHypInvasiveDebugEnabled() == TRUE.

Other values are reserved.
For ARMv8-R:
- If EL2 is implemented, bit [9] is RAO.
- If EL2 is not implemented, bits [9:8] are RAZ.
SNID, bits [7:6]
Secure non-invasive debug. Possible values of this field are:
00 Not implemented.
Other values are reserved.

SID, bits [5:4]
Secure invasive debug. Possible values of this field are:
00 Not implemented.
Other values are reserved.

NSNID, bits [3:2]
Non-secure non-invasive debug. Possible values of this field are:
10 Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
11 Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.
Other values are reserved.

NSID, bits [1:0]
Non-secure invasive debug. Possible values of this field are:
10 Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
11 Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.
Other values are reserved.

Accessing the DBGAUTHSTATUS
This register can be read using MRC with the following syntax:
MRC <syntax>
This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p14, 0, &lt;Rt&gt;, c7, c14, 6</td>
<td>000</td>
<td>110</td>
<td>0111</td>
<td>1110</td>
<td>1110</td>
</tr>
</tbody>
</table>

Accessibility
The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p14, 0, &lt;Rt&gt;, c7, c14, 6</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p14, 0, &lt;Rt&gt;, c7, c14, 6</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables
For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
When EL2 is implemented:

- If $\text{HDCR.TDA}==1$, read accesses to this register from EL1 are trapped to Hyp mode.
### E2.1.2 DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

**Purpose**

Main control register for the debug implementation.

**Configurations**

This register is required in all implementations.

This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset RW fields in this register reset to architecturally **UNKNOWN** values.

**Attributes**

DBGDSCRext is a 32-bit register.

**Field descriptions**

The DBGDSCRext bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>RXfull, bit [30]</td>
</tr>
<tr>
<td>29</td>
<td>TXfull, bit [29]</td>
</tr>
<tr>
<td>28</td>
<td>RES0</td>
</tr>
<tr>
<td>27</td>
<td>RXO</td>
</tr>
<tr>
<td>26</td>
<td>TXU</td>
</tr>
<tr>
<td>25</td>
<td>RES0</td>
</tr>
<tr>
<td>24</td>
<td>INTdis</td>
</tr>
<tr>
<td>23</td>
<td>TDA</td>
</tr>
<tr>
<td>22</td>
<td>RES0</td>
</tr>
<tr>
<td>21</td>
<td>SPNIDdis</td>
</tr>
<tr>
<td>20</td>
<td>SPIDdis</td>
</tr>
<tr>
<td>19</td>
<td>INTdis</td>
</tr>
<tr>
<td>18</td>
<td>TDA</td>
</tr>
<tr>
<td>17</td>
<td>RES0</td>
</tr>
<tr>
<td>16</td>
<td>SPNIDdis</td>
</tr>
<tr>
<td>15</td>
<td>SPIDdis</td>
</tr>
<tr>
<td>14</td>
<td>HDE</td>
</tr>
<tr>
<td>13</td>
<td>UDCCdis</td>
</tr>
<tr>
<td>12</td>
<td>ERR</td>
</tr>
<tr>
<td>11</td>
<td>RES0</td>
</tr>
<tr>
<td>10</td>
<td>MOE</td>
</tr>
<tr>
<td>9</td>
<td>RES0</td>
</tr>
<tr>
<td>8</td>
<td>RES0</td>
</tr>
<tr>
<td>7</td>
<td>RES0</td>
</tr>
<tr>
<td>6</td>
<td>RES0</td>
</tr>
<tr>
<td>5</td>
<td>RES0</td>
</tr>
<tr>
<td>4</td>
<td>RES0</td>
</tr>
<tr>
<td>3</td>
<td>RES0</td>
</tr>
<tr>
<td>2</td>
<td>RES0</td>
</tr>
<tr>
<td>1</td>
<td>RES0</td>
</tr>
<tr>
<td>0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**RXfull, bit [30]**

DTRRX full. Used for save/restore of **EDSCR.RXfull**.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of **EDSCR.RXfull**.

ARM deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRRX full status.

The architected behavior of this field determines the value it returns after a reset.

**TXfull, bit [29]**

DTRTX full. Used for save/restore of **EDSCR.TXfull**.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of EDSCR.TXfull.
ARM deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRTX full status.
The architected behavior of this field determines the value it returns after a reset.

Bit [28]
Reserved, RES0.

RXO, bit [27]
Used for save/restore of EDSCR.RXO.
When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of EDSCR.RXO.
The architected behavior of this field determines the value it returns after a reset.

TXU, bit [26]
Used for save/restore of EDSCR.TXU.
When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of EDSCR.TXU.
The architected behavior of this field determines the value it returns after a reset.

Bits [25:24]
Reserved, RES0.

INTdis, bits [23:22]
Used for save/restore of EDSCR.INTdis.
When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this field is RO, and software must treat it as UNK/SBZP.
When DBGOSLSR.OSLK == 1 (the OS lock is locked), this field is RW and holds the value of EDSCR.INTdis.
The architected behavior of this field determines the value it returns after a reset.

TDA, bit [21]
Used for save/restore of EDSCR.TDA.
When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of EDSCR.TDA.
The architected behavior of this field determines the value it returns after a reset.

Bits [20:19]
Reserved, RES0.

NS, bit [18]
Non-secure status. This bit is RES1.
ARM deprecates use of this field.

SPNIDdis, bit [17]
Secure privileged profiling disabled status bit. This bit is RES0.
ARM deprecates use of this field.

**SPIDdis, bit [16]**

Secure privileged AArch32 invasive self-hosted debug disabled status bit. This bit is RES0. ARM deprecates use of this field.

**MDBGen, bit [15]**

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

- 0Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
- 1Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

When this register has an architecturally defined reset value, this field resets to 0.

**HDE, bit [14]**

Used for save/restore of EDSCR.HDE.

- When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
- When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of EDSCR.HDE.

The architected behavior of this field determines the value it returns after a reset.

**Bit [13]**

Reserved, RES0.

**UDCCdis, bit [12]**

Traps EL0 accesses to the DCC registers to Undefined mode.

- 0EL0 accesses to the DCC registers are not trapped to Undefined mode.
- 1EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, and DBGDRAR are trapped to Undefined mode.

--- Note

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRR accesses to DBGDSAR and DBGDRAR.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

When this register has an architecturally defined reset value, this field resets to 0.

**Bits [11:7]**

Reserved, RES0.

**ERR, bit [6]**

Used for save/restore of EDSCR.ERR.

- When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.
- When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW and holds the value of EDSCR.ERR.

The architected behavior of this field determines the value it returns after a reset.

**MOE, bits [5:2]**

Method of Entry for debug exception. When a debug exception is taken, this field is set to indicate the event that caused the exception:

- 0001Breakpoint
- 0011Software breakpoint (BKPT) instruction
- 0101Vector catch
1010 Watchpoint

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [1:0]

Reserved, RES0.

Accessing the DBGDSCRext

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p14, 0, &lt;Rt&gt;, c0, c2, 2</td>
<td>000</td>
<td>010</td>
<td>0000</td>
<td>1110</td>
<td>0010</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p14, 0, &lt;Rt&gt;, c0, c2, 2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p14, 0, &lt;Rt&gt;, c0, c2, 2</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Individual fields within this register might have restricted accessibility when DBGOSLSR.OSLK == 0 (the OS lock is unlocked.) See the field descriptions for more detail.

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HDCR.TDA==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.3 DFSR, Data Fault Status Register

The DFSR characteristics are:

**Purpose**
Holds status information about the last data fault.

**Configurations**
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**
DFSR is a 32-bit register.

**Field descriptions**
The DFSR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>16</td>
<td>FnV, bit [16]</td>
</tr>
<tr>
<td>12</td>
<td>ExT, bit [12]</td>
</tr>
<tr>
<td>13</td>
<td>CM, bit [13]</td>
</tr>
<tr>
<td></td>
<td>Cache maintenance fault. For synchronous faults, this bit indicates whether</td>
</tr>
<tr>
<td></td>
<td>a cache maintenance instruction generated the fault. The possible values of</td>
</tr>
<tr>
<td></td>
<td>this bit are:</td>
</tr>
<tr>
<td>1</td>
<td>Abort caused by execution of a cache maintenance instruction.</td>
</tr>
<tr>
<td>0</td>
<td>Abort not caused by execution of a cache maintenance instruction.</td>
</tr>
<tr>
<td>6</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>9</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>10</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>11</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>14</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>15</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>17</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>18</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>19</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>21</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>22</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>23</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>24</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>25</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>26</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>27</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>28</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>29</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

This field is only valid for a Synchronous external abort. It is RES0 for all other Data Abort exceptions.

**FnV, bit [16]**
FAR not Valid, for a Synchronous external abort.
0  DFAR is valid.
1  DFAR is not valid, and holds an UNKNOWN value.

**Bits [31:17]**
Reserved, RES0.

**Bits [15:14]**
Reserved, RES0.

**CM, bit [13]**
Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault. The possible values of this bit are:
0  Abort not caused by execution of a cache maintenance instruction.
1  Abort caused by execution of a cache maintenance instruction.

On an asynchronous fault, this bit is UNKNOWN.

**ExT, bit [12]**
External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.
In an implementation that does not provide any classification of external aborts, this bit is RES0.
For aborts other than external aborts this bit always returns 0.
Write not Read bit. Indicates whether the abort was caused by a write or a read instruction. The possible values of this bit are:
0 Abort caused by a read instruction.
1 Abort caused by a write instruction.
For faults on the cache maintenance and address translation System instructions in the (coproc==1111) encoding space this bit always returns a value of 1.

Bit [10]
Reserved, RES0.

LPAE, bit [9]
Reserved, RES1.

Bits [8:6]
Reserved, RES0.

STATUS, bits [5:0]
Fault status bits. Possible values of this field are:
000100 Translation fault
001100 Permission fault
010000 Synchronous external abort, other than synchronous parity or ECC error
010001 SError interrupt
011000 Synchronous parity or ECC error on memory access
011001 SError parity or ECC error on memory access
100001 Alignment fault
100010 Debug exception
110100 IMPLEMENTATION DEFINED fault (Cache lockdown fault)
110101 IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault)
All other values are reserved.

Accessing the DFSR
This register can be read using MRC with the following syntax:
MRC <syntax>
This register can be written using MCR with the following syntax:
MCR <syntax>
This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;R&gt;, c5, c0, 0</td>
<td>000</td>
<td>000</td>
<td>0101</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>
Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c5, c0, 0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c5, c0, 0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T5==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.4 HCPTR, Hyp Architectural Feature Trap Register

The HCPTR characteristics are:

**Purpose**

Controls:
- Trapping to Hyp mode of access, at EL1 or EL0, to trace, and to Advanced SIMD or floating-point functionality.
- Hyp mode access to trace, and to Advanced SIMD or floating-point functionality.

--- **Note** ---

Accesses to this functionality:
- Other than Hyp mode are also affected by settings in the CPACR and NSACR.
- From Hyp mode are also affected by settings in the NSACR.

Exceptions generated by the CPACR and NSACR controls are higher priority than those generated by the HCPTR controls.

**Usage constraints**

This register is accessible as follows:

<table>
<thead>
<tr>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RW</td>
</tr>
</tbody>
</table>

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Subject to the prioritization rules:

- If HSTR.T1=1, accesses to this register from EL1 are trapped to Hyp mode.

**Configurations**

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2. Otherwise, RW fields in this register reset to architecturally **UNKNOWN** values.

**Attributes**

HCPTR is a 32-bit register.

**Field descriptions**

The HCPTR bit assignments are:
TCPAC, bit [31]
Traps EL1 accesses to the CPACR to Hyp mode.
0 This control has no effect on EL1 accesses to the CPACR.
1 EL1 accesses to the CPACR are trapped to Hyp mode.

--- Note ---
The CPACR is not accessible at EL0.

When this register has an architecturally-defined reset value, this field resets to 0.

Bits [30:21]
Reserved, RES0.

TTA, bit [20]
Traps System register accesses to all implemented trace registers to Hyp mode.
0 This control has no effect on System register accesses to trace registers.
1 Any System register access to an implemented trace register is trapped to Hyp mode, unless the access is trapped to EL1 by a CPACR or NSACR control, or the access is from EL0 and the definition of the register in the appropriate trace architecture specification indicates that the register is not accessible from EL0. A trapped instruction generates:
  • A Hyp Trap exception, if the exception is taken from EL0 or EL1.
  • An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

If the implementation does not include a trace macrocell, or does not include a System register interface to the trace macrocell registers, it is IMPLEMENTATION DEFINED whether this bit:
  • Is RES0.
  • Is RES1.
  • Can be written from Hyp mode.

--- Note ---
• The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and a resulting Undefined Instruction exception is higher priority than a HCPTR.TTA Hyp Trap exception.
• The architecture does not provide traps on trace register accesses through the optional memory-mapped external debug 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.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

Bits [19:16]
Reserved, RES0.

TASE, bit [15]
Traps execution of Advanced SIMD instructions to Hyp mode when the value of HCPTR.TCP10 is 0.
0 This control has no effect on execution of Advanced SIMD instructions.
When the value of HCPTR.TCP10 is 0, any attempt to execute an Advanced SIMD instruction is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from NEL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

When the value of HCPTR.TCP10 is 1, the value of this field is ignored.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1. If the implementation includes floating-point functionality but does not include Advanced SIMD functionality, this field is RES1. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, then it is RAZ/WI.

For the list of instructions affected by this field, see "Controls of Advanced SIMD operation that do not apply to floating-point operation" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

**Bit [14]**

Reserved, RES0.

**Bits [13:12]**

Reserved, RES1.

**TCP11, bit [11]**

The value of this field is ignored. If this field is programmed with a different value to the TCP10 bit then this field is UNKNOWN on a direct read of the HCPTR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

**TCP10, bit [10]**

Trap accesses to Advanced SIMD and floating-point functionality to Hyp mode:

0  This control has no effect on accesses to Advanced SIMD and floating-point functionality.

1  Any attempted access to Advanced SIMD and floating-point functionality is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.
Bits [9:0]

Reserved, RES1.

Accessing the HCPTR:

To access the HCPTR:

MRC p15,4,<Rt>,c1,c1,2 ; Read HCPTR into Rt
MCR p15,4,<Rt>,c1,c1,2 ; Write Rt to HCPTR

Register access is encoded as follows:

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>010</td>
</tr>
</tbody>
</table>
E2.1.5 HCR, Hyp Configuration Register

The HCR characteristics are:

**Purpose**

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to Hyp mode.

**Configurations**

Some or all RW fields of this register have defined reset values. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HCR is a 32-bit register.

**Field descriptions**

The HCR bit assignments are:

```
   31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
   \-----------------------------------------------\-----------------------------------------------
   | RES0 | TRVM | HCD | RES0 | TGE | TVM | RES0 | TPU | TPC | TSW | TAC | TIDCP | RES0 | TID3 | TID2 | TID1 | BSU | FB | VI |
   \-----------------------------------------------\-----------------------------------------------
   | VM   | SWIO | RES0 | FMO | IMO | AMO | VF   | VA  | DC  | TWI | TWE | TID0 |
   \-----------------------------------------------\-----------------------------------------------
```

**Bit [31]**

Reserved, RES0.

**TRVM, bit [30]**

Trap Reads of Memory controls. Traps EL1 reads of the memory control registers to Hyp mode. The registers for which read accesses are trapped are as follows:

- SCTLR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFS, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR, PRSLR, PRBAR, PRBAR<n>, PRLAR, PRLAR<n>.

0  This control has no effect on EL1 read accesses to memory control registers.

1  EL1 read accesses to the specified memory control registers are trapped to Hyp mode.

When this register has an architecturally defined reset value, this field resets to 0.
HCD, bit [29]

HVC instruction disable. Disables execution of HVC instructions.

0  HVC instruction execution is enabled at EL2 and EL1.
1  HVC instructions are UNDEFINED at EL2 and EL1. The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

--- Note ---

HVC instructions are always UNDEFINED at EL0.

Note: When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

Bit [28]

Reserved, RES0.

TGE, bit [27]

Trap General Exceptions from EL0.

0  This control has no effect on execution at EL0.
1  All exceptions that would be routed to EL1 are routed to EL2.
   The SCTLR.M bit is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR.
   The HCR.{FMO, IMO, AMO} bits are treated as being 1 for all purposes other than returning the result of a direct read of HCR.
   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 HDCR.{TDRA, TDOSA, TDA} bits are ignored and treated as being 1 other than for the purpose of a direct read of HDCR.

When this register has an architecturally defined reset value, this field resets to 0.

Bit [26]

Trap Memory controls. Traps EL1 writes to the memory control registers to Hyp mode. The registers for which write accesses are trapped are as follows:

SCTLR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR, PRSELR, PRBAR, PRBAR<n>, PRLAR, PRLAR<n>.

0  This control has no effect on EL1 write accesses to EL1 memory control registers.
1  EL1 write accesses to EL1 memory control registers are trapped to Hyp mode.

When this register has an architecturally defined reset value, this field resets to 0.

Bit [25]

Reserved, RES0.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps EL1 execution of those cache maintenance instructions to Hyp mode. This applies to the following instructions:

ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU.

--- Note ---

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0  This control has no effect on the execution of cache maintenance instructions.
EL1 execution of the specified instructions is trapped to Hyp mode. When this register has an architecturally defined reset value, this field resets to 0.

**TPC, bit [23]**
Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps EL1 execution of those cache maintenance instructions to Hyp mode. This applies to the following instructions:
DCIMVAC, DCCIMVAC, DCCMVAC.

--- **Note** ---
An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0 This control has no effect on the execution of cache maintenance instructions.
1 EL1 execution of the specified instructions is trapped to Hyp mode. When this register has an architecturally defined reset value, this field resets to 0.

**TSW, bit [22]**
Trap data or unified cache maintenance instructions that operate by Set/Way. Traps EL1 execution of those cache maintenance instructions by set/way to Hyp mode. This applies to the following instructions:
DCISW, DCCSW, DCCISW.

--- **Note** ---
An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0 This control has no effect on the execution of cache maintenance instructions.
1 EL1 execution of the specified instructions is trapped to Hyp mode. When this register has an architecturally defined reset value, this field resets to 0.

**TAC, bit [21]**
Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to Hyp mode. This applies to the following register accesses:
ACTLR and, if implemented, ACTLR2.

0 This control has no effect on EL1 accesses to the Auxiliary Control Registers.
1 EL1 accesses to the specified registers are trapped to Hyp mode. When this register has an architecturally defined reset value, this field resets to 0.

**TIDCP, bit [20]**
Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings for IMPLEMENTATION DEFINED System Registers to Hyp mode.

MCR and MRC instructions accessing the following encodings:
- All coproc==p15, CRn==c9, Opcode1 = {0-7}, CRm = {c0-c2, c5-c8}, opcode2 = {0-7}.
- All coproc==p15, CRn==c10, Opcode1 = {0-7}, CRm = {c0, c1, c4, c8}, opcode2 = {0-7}.
- All coproc==p15, CRn==c11, Opcode1 = {0-7}, CRm = {c0-c8, c15}, opcode2 = {0-7}.

When HCR.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from EL0 is trapped to Hyp mode. If it is not, it is UNDEFINED, and the PE takes an Undefined Instruction exception to Undefined mode.

0 This control has no effect on EL1 and EL0 accesses to the System register encodings for IMPLEMENTATION DEFINED functionality.
1. EL1 accesses to the specified System register encodings for IMPLEMENTATION DEFINED functionality are trapped to Hyp mode.

When this register has an architecturally defined reset value, this field resets to 0.

**Bit [19]**

Reserved, RES0.

**TID3, bit [18]**

Trap ID group 3. Traps EL1 reads of the following registers to Hyp mode:

ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, MVFR0, MVFR1, MVFR2, and ID_MMFR4, except that if ID_MMFR4 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 are trapped.

Also an MRC access to any of the following encodings:

- coproc==p15, opc1 == 0, CRn == c0, CRm == {c3-c7}, opc2 == {0,1}.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c3, opc2 == 2.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c5, opc2 == {4,5}.

It is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to the following encodings:

- coproc==p15, opc1 == 0, CRn == c0, CRm == c2, opc2 == 7.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c3, opc2 == {3-7}.
- coproc==p15, opc1 == 0, CRn == c0, CRm == {c4, c6, c7}, opc2 == {2-7}.
- coproc==p15, opc1 == 0, CRn == c0, CRm == c5, opc2 == {2, 3, 6, 7}.

0. This control has no effect on EL1 reads of the ID group 3 registers.
1. The specified EL1 read accesses to ID group 3 registers are trapped to Hyp mode.

When this register has an architecturally defined reset value, this field resets to 0.

**TID2, bit [17]**

Trap ID group 2. Traps the following register accesses to Hyp mode:

- EL1 and EL0 reads of the CTR, CCSIDR, CLIDR, and CSSELR.
- EL1 and EL0 writes to the CSSELR.

0. This control has no effect on EL1 and EL0 accesses to the ID group 2 registers.
1. The specified EL1 and EL0 accesses to ID group 2 registers are trapped to Hyp mode.

When this register has an architecturally defined reset value, this field resets to 0.

**TID1, bit [16]**

Trap ID group 1. Traps EL1 reads of the following registers to Hyp mode:

TCMTR, TLBTR, REVIDR, AIDR, MPUIR.

0. This control has no effect on EL1 reads of the ID group 1 registers.
1. The specified EL1 read accesses to ID group 1 registers are trapped to Hyp mode.

When this register has an architecturally defined reset value, this field resets to 0.

**TID0, bit [15]**

Trap ID group 0. Traps the following register accesses to Hyp mode:

- EL1 reads of the JIDR and FPSID.
- If the JIDR is RAZ from EL0, EL0 reads of the JIDR.

--- **Note** ---

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
- The FPSID is not accessible at EL0.
- Writes to the FPSID are ignored, and not trapped by this control.

\[\text{0} \quad \text{This control has no effect on EL1 reads of the ID group 0 registers.}\]
\[\text{1} \quad \text{The specified EL1 read accesses to ID group 0 registers are trapped to Hyp mode.}\]

When this register has an architecturally defined reset value, this field resets to 0.

**TWE, bit [14]**

Traps EL0 and EL1 execution of WFE instructions to Hyp mode:

\[\text{0} \quad \text{This control has no effect on the execution of WFE instructions at EL0 or EL1.}\]
\[\text{1} \quad \text{Any attempt to execute a WFE instruction at EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR.nTWE is 0, the trap of EL0 execution to Undefined mode takes precedence over this trap.}\]

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

--- 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 this register has an architecturally defined reset value, this field resets to 0.

**TWI, bit [13]**

Traps EL0 and EL1 execution of WFI instructions to Hyp mode.

\[\text{0} \quad \text{This control has no effect on the execution of WFI instructions at EL1 or EL0.}\]
\[\text{1} \quad \text{Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR.nTWI is 0, the trap of EL0 execution to Undefined mode takes precedence over this trap.}\]

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

--- 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 this register has an architecturally defined reset value, this field resets to 0.

**DC, bit [12]**

Default Cacheability.

\[\text{0} \quad \text{This control has no effect on the EL1&0 translation regime.}\]
\[\text{1} \quad \text{The HCR.VM field behaves as 1 for all purposes other than a direct read of the value of the field.}\]

The memory type produced by the first stage default memory map of the EL1&0 translation regime is:

- Normal memory.
- Non-shareable.
- Inner Write-Back Read-Allocate, Write-Allocate Cacheable.
- Outer Write-Back Read-Allocate, Write-Allocate Cacheable.
For instruction accesses, not XN, meaning that execution is permitted.
This field has no effect on the EL2 translation regime.
When this register has an architecturally defined reset value, this field resets to 0.

**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:

- 00: No effect
- 01: Inner Shareable
- 10: Outer Shareable
- 11: Full system

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the Shareability attributes from two stages of address translation.
When this register has an architecturally defined reset value, this field resets to 0.

**FB, bit [9]**
Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from EL1:
BPIALL, ICIALLU.

- 0: This field has no effect on the operation of the specified instructions.
- 1: When one of the specified instruction is executed at EL1, the instruction is broadcast within the Inner Shareable Shareability domain.

When this register has an architecturally defined reset value, this field resets to 0.

**VA, bit [8]**
Virtual SError interrupt exception.

- 0: This mechanism is not making a virtual SError interrupt pending.
- 1: A virtual SError interrupt is pending because of this mechanism.

The virtual SError interrupt is only enabled when the value of HCR.AMO bit is 1.
The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.
When this register has an architecturally defined reset value, this field resets to 0.

**VI, bit [7]**
Virtual IRQ exception.

- 0: This mechanism is not making a virtual IRQ pending.
- 1: A virtual IRQ is pending because of this mechanism.

The virtual IRQ is only enabled when the value of HCR.IMO bit is 1.
The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.
When this register has an architecturally defined reset value, this field resets to 0.

**VF, bit [6]**
Virtual FIQ exception.

- 0: This mechanism is not making a virtual FIQ pending.
- 1: A virtual FIQ is pending because of this mechanism.

The virtual FIQ is only enabled when the value of HCR.FMO bit is 1.
The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.
When this register has an architecturally defined reset value, this field resets to 0.
AMO, bit [5]
SError interrupt Mask Override. When this bit is set to 1, it overrides the effect of CPSR.A, and enables virtual exception signaling by the VA bit.
If the value of HCR.TGE is 1, then the HCR.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.
When this register has an architecturally defined reset value, this field resets to 0.

IMO, bit [4]
IRQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.I, and enables virtual exception signaling by the VI bit.
If the value of HCR.TGE is 1, then the HCR.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.
When this register has an architecturally defined reset value, this field resets to 0.

FMO, bit [3]
FIQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.F, and enables virtual exception signaling by the VF bit.
If the value of HCR.TGE is 1, then the HCR.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.
When this register has an architecturally defined reset value, this field resets to 0.

Bit [2]
Reserved, RES0.

SWIO, bit [1]
Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to be treated as data cache clean and invalidate by set/way.
0 This control has no effect on the operation of data cache invalidate by set/way instructions.
1 Data cache invalidate by set/way instructions operate as data cache clean and invalidate by set/way.
When this bit is set to 1, DCISW is executed as DCCISW.
As a result of changes to the behavior of DCISW, this bit is redundant in ARMv8. This bit can be implemented as RES1.
When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

VM, bit [0]
Virtualization enable. Enables stage 2 protection for EL1 and EL0 accesses. This is provided by the EL2-controlled MPU. Possible values of this bit are:
0 EL1 and EL0 stage 2 address protection disabled.
1 EL1 and EL0 stage 2 address protection enabled and attribute combination enabled.
If the HCR.DC bit is set to 1, then the behavior of the PE when executing in a mode other than Hyp mode is consistent with HCR.VM being 1, regardless of the actual value of HCR.VM, other than the value returned by an explicit read of HCR.VM.
When the value of this bit is 1, data cache invalidate instructions executed at EL1 operate as data cache clean and invalidate instructions. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR.SWIO bit.
When this register has an architecturally defined reset value, this field resets to 0.

Accessing the HCR
This register can be read using MRC with the following syntax:
E2 Description of the Redefined or New System Registers
E2.1 Redefined System registers

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c1, 0</td>
<td>100</td>
<td>000</td>
<td>0001</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c1, 0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c1, 0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T1==1, accesses to this register from EL1 are trapped to Hyp mode.
### E2.1.6 HCR2, Hyp Configuration Register 2

The HCR2 characteristics are:

**Purpose**

Provides additional configuration controls for virtualization.

**Configurations**

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HCR2 is a 32-bit register.

**Field descriptions**

The HCR2 bit assignments are:

<table>
<thead>
<tr>
<th>Bit assignments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:7</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>6</td>
<td>MIOCNCE, bit [6]</td>
</tr>
</tbody>
</table>

**Bits [31:7]**

Reserved, RES0.

**MIOCNCE, bit [6]**

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

0 For the EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

1 For the EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information see "Mismatched memory attributes" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The value of this field has no effect on translation regimes other than the EL1&0 translation regime.

This field can be implemented as RAZ/WI.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

**Bits [5:0]**

Reserved, RES0.

**Accessing the HCR2**

This register can be read using MRC with the following syntax:

```
MRC <syntax>
```

This register can be written using MCR with the following syntax:

```
MCR <syntax>
```
This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, cl, c1, 4</td>
<td>100</td>
<td>100</td>
<td>0001</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

### Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, cl, c1, 4</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, cl, c1, 4</td>
<td>1</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state” in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T1==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.7 **HDCR, Hyp Debug Control Register**

The HDCR characteristics are:

**Purpose**

Controls the trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors Extension.

**Configurations**

This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset RW fields in this register reset to architecturally **UNKNOWN** values.

**Attributes**

HDCR is a 32-bit register.

**Field descriptions**

The HDCR bit assignments are:

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-22</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>21</td>
<td><strong>EPMAD</strong>, bit [21]</td>
</tr>
<tr>
<td>19-18</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>17</td>
<td><strong>HPMD</strong>, bit [17]</td>
</tr>
<tr>
<td>16-12</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>11-8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7</td>
<td>Access to Performance Monitors counters by an external debugger is disabled, unless overridden by ExternalHypNoninvasiveDebugEnabled().</td>
</tr>
<tr>
<td>6</td>
<td>Event counting by EL1-accessible counters allowed in Hyp mode.</td>
</tr>
<tr>
<td>5</td>
<td>Event counting by EL1-accessible counters prohibited in Hyp mode.</td>
</tr>
</tbody>
</table>

**Bits [31:22]**

Reserved, RES0.

**EPMAD, bit [21]**

External debug interface access to Hyp mode Performance Monitors registers disable. This disables access by an external debugger to counters that are reserved for use from Hyp mode:

- **0**: Access to all Performance Monitors counters by an external debugger is permitted.
- **1**: Access to Performance Monitors counters in the range [HPMN..(PMCR.N-1)] by an external debugger is disabled, unless overridden by ExternalHypNoninvasiveDebugEnabled().

If the Performance Monitors Extension is not implemented or does not support external debug interface accesses this bit is RES0.

When this register has an architecturally defined reset value, this field resets to 0.

**Bits [20:18]**

Reserved, RES0.

**HPMD, bit [17]**

Hyp Performance Monitors Disable. This control prohibits event counting in Hyp mode by the counters accessible at EL1.

- **0**: Event counting by EL1-accessible counters allowed in Hyp mode.
- **1**: Event counting by EL1-accessible counters prohibited in Hyp mode.
This control applies only to:

- The event counters in the range [0..HPMN-1].
- If PMCR.DP is set to 1, PMCCNTR.

The other event counters are unaffected. When PMCR.DP is set to 0, PMCCNTR is unaffected.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally defined reset value, this field resets to 0.

### Bits [16:12]

Reserved, RES0.

### TDRA, bit [11]

Trap Debug ROM Address register access. Traps EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode.

- 0 EL0 and EL1 System register accesses to the Debug ROM registers are not trapped to Hyp mode.
- 1 EL0 and EL1 System register accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally defined reset value, this field resets to 0.

### TDOSA, bit [10]

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to Hyp mode.

- 0 EL1 System register accesses to the powerdown debug registers are not trapped to Hyp mode.
- 1 EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, DBGOSDLR, and the DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

#### Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally defined reset value, this field resets to 0.

### TDA, bit [9]

Trap debug access. Traps EL0 and EL1 System register accesses to those debug System registers in the (coproc==1110) encoding space that are not trapped by either of the following:

- HDCR.TDRA.
- HDCR.TDOSA.

- 0 Has no effect on System register accesses to the debug registers.
- 1 EL0 or EL1 System register accesses to the debug registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA, are trapped to Hyp mode.

Traps of accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.
When this register has an architecturally defined reset value, this field resets to 0.

**TDE, bit [8]**

Trap Debug exceptions. The possible values of this bit are:

0  This control has no effect on the routing of debug exceptions, and has no effect on accesses to debug registers.
1  Debug exceptions generated at EL1 or EL0 are routed to EL2.

All accesses to Debug registers that would not be UNDEFINED if the value of this field was 0 are trapped to EL2.

When `HCR.TGE == 1`, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

When this register has an architecturally defined reset value, this field resets to 0.

**HPME, bit [7]**

Hyp Performance Monitors Enable. The possible values of this bit are:

0  Hyp Performance Monitors disabled.
1  Hyp Performance Monitors enabled.

When the value of this bit is 1, the Performance Monitors counters that are reserved for use from Hyp mode are enabled. For more information see the description of the HPMN field.

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

**TPM, bit [6]**

Trap Performance Monitors accesses. Traps EL0 and EL1 accesses to all Performance Monitors registers to Hyp mode.

0  EL0 and EL1 accesses to all Performance Monitors registers are not trapped to Hyp mode.
1  EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

--- **Note** ---

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

---

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

**TPMCR, bit [5]**

Trap PMCR accesses. Traps EL0 and EL1 accesses to the PMCR to Hyp mode.

0  EL0 and EL1 accesses to the PMCR are not trapped to Hyp mode.
1  EL0 and EL1 accesses to the PMCR are trapped to Hyp mode.

--- **Note** ---

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

---

If the Performance Monitors Extension is not implemented, this field is RES0.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.
HPMN, bits [4:0]

Defines the number of Performance Monitors counters that are accessible from EL1 modes, and from EL0 modes if unprivileged access is enabled.

If the Performance Monitors Extension is not implemented, this field is RES0.

HPMN divides the Performance Monitors counters as follows. If software is accessing Performance Monitors counter n then:

- If \( n \) is in the range \( 0 \leq n < \text{HPMN} \), the counter is accessible from EL1 and EL2, and from EL0 if unprivileged access to the counters is enabled. PMCR.E enables the operation of counters in this range. HDCR.HPMD disables the operation of counters in this range in EL2. The counters in this range are always accessible from the IMPLEMENTATION DEFINED external debug interface.

- If \( n \) is in the range \( \text{HPMN} \leq n < \text{PMCR.N} \), the counter is accessible only from EL2. HDCR.HPME enables the operation of counters in this range.

If this field is set to 0, or to a value larger than PMCR.N, then the following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN non-zero value less than PMCR.N.
  - All counters are reserved for EL2 use, meaning no counters are accessible from EL1 and EL0.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to the value of PMCR.N.

Accessing the HDCR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>Syntax (p15, 4, &lt;Rt&gt;, c1, c1, l)</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c1, l</td>
<td>100</td>
<td>001</td>
<td>0001</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>Syntax (p15, 4, &lt;Rt&gt;, c1, c1, l)</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c1, l</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c1, l</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T1==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.8 HSCTLR, Hyp System Control Register

The HSCTLR characteristics are:

**Purpose**

Provides top-level control of the system operation in Hyp mode.

**Configurations**

Some or all RW fields of this register have defined reset values. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HSCTLR is a 32-bit register.

**Field descriptions**

The HSCTLR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>TE, bit. T32 Exception Enable. This bit controls whether exceptions to EL2 are taken to A32 or T32 state:</td>
</tr>
<tr>
<td>29</td>
<td>0 Exceptions, including reset, taken to A32 state.</td>
</tr>
<tr>
<td>28</td>
<td>1 Exceptions, including reset, taken to T32 state.</td>
</tr>
<tr>
<td>27-26</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>25</td>
<td>EE, bit. The value of the PSTATE.E bit on entry to Hyp mode.</td>
</tr>
<tr>
<td>24</td>
<td>The possible values of this bit are:</td>
</tr>
<tr>
<td>23-22</td>
<td>0 Little-endian. PSTATE.E is cleared to 0 on entry to Hyp mode.</td>
</tr>
<tr>
<td>21</td>
<td>1 Big-endian. PSTATE.E is set to 1 on entry to Hyp mode.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>19</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>18</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>17</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>16</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>15</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>14</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>13</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>12</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>11</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>10</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>9</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>6</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>5</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>4</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>2</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>1</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
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.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

FI, bit [21]

Fast Interrupts enable. Permitted values of this bit are:

0 All performance features enabled.

1 Low interrupt latency configuration. Some performance features disabled.

In ARMv8-R:

- If FI == 1, then asynchronous errors and interrupts can interrupt LDM/STM instructions that access Normal memory. The register that provides the base address is restored to its original value and any other modified registers become UNKNOWN.

- Up to two MPUs can determine whether an access is to Normal memory, therefore the value of the FI field depends on whether the appropriate MPU or MPUs allow the access to the location as access to Normal memory.

  - For EL0 and EL1 accesses, this is the combination of the attributes determined by the configuration settings of the EL1-controlled MPU and the configuration settings of the EL2-controlled MPU.

  - For EL2 accesses, these are the attributes determined by the configuration settings of the EL2-controlled MPU.

When this register has an architecturally defined reset value, this field resets to 0.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [18]

Reserved, RES1.

BR, bit [17]

Background Region enable for EL2.

0 EL2-controlled MPU Background region disabled. Any EL2 transaction that does not match an EL2-controlled MPU region results in a fault.

1 EL2-controlled MPU Background region enabled. For EL2 transactions that do not match an EL2-controlled MPU region, the EL2 Background region attributes are used.

When this register has an architecturally defined reset value, this field resets to 0.
Bit [16]
Reserved, RES1.

Bits [15:13]
Reserved, RES0.

I, bit [12]
Instruction access Cacheability control, for accesses at EL2:
0 All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.
If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
1 All instruction access to Normal memory from EL2 can be cached at all levels of instruction and unified cache.
If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.
This bit has no effect on the EL1&0 translation regime.
When this register has an architecturally defined reset value, this field resets to 0.

Bit [11]
Reserved, RES1.

Bits [10:9]
Reserved, RES0.

SED, bit [8]
SETEND instruction disable. Disables SETEND instructions at EL2.
0 SETEND instruction execution is enabled at EL2.
1 SETEND instructions are UNDEFINED at EL2.
If the implementation does not support mixed-endian operation at EL2, this bit is RES1.
When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

ITD, bit [7]
IT Disable. Disables some uses of IT instructions at EL2.
0 All IT instruction functionality is enabled at EL2.
1 Any attempt at EL2 to execute any of the following is UNDEFINED:
• All encodings of the IT instruction with hw1[3:0]!-1000.
• All encodings of the subsequent instruction with the following values for hw1:
  11xxxxxxxxxxxx
  All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
  1011xxxxxxxxxx
  All instructions in “Miscellaneous 16-bit instructions” in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
  10100xxxxxxxx
  ADD Rd, PC, #imm
  01001xxxxxxxxx
  LDR Rd, [PC, #imm]
0100x1xxx1111xxx
ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
010001xx1xxxx111
ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

• A 16-bit instruction, that can only be followed by another 16-bit instruction.
• The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

ITD is optional, but if it is implemented in the SCTL then it must also be implemented in the HSCTL. If it is not implemented then this bit is RAZ/WI.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bit [6]
Reserved, RES0.

CP15BEN, bit [5]
System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==1111) encoding space from EL2:

0  EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
1  EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTL then it must also be implemented in the HSCTL. If it is not implemented then this bit is RAO/WI.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

Bits [4:3]
Reserved, RES1.

C, bit [2]
Cacheability control, for data accesses at EL2:

0  All data accesses to Normal memory from EL2 are Non-cacheable for all levels of data and unified cache.
1  All data accesses to Normal memory from EL2 can be cached at all levels of data and unified cache.

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

When this register has an architecturally defined reset value, this field resets to 0.

A, bit [1]
Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

0  Alignment fault checking disabled when executing at EL2. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element or data elements being accessed.
1  Alignment fault checking enabled when executing at EL2.
All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element or data elements being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

M, bit [0]

MPU enable for the EL2-controlled MPU. Possible values of this bit are:

0 EL2-controlled MPU disabled.

1 EL2-controlled MPU enabled.

When this register has an architecturally defined reset value, this field resets to 0.

Accessing the HSCTLR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c0, 0</td>
<td>100</td>
<td>000</td>
<td>0001</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c0, 0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c1, c0, 0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

• If HSTR.T1==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.9 HSR, Hyp Syndrome Register

The HSR characteristics are:

Purpose

Holds syndrome information for an exception taken to Hyp mode.

Configurations

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

HSR is a 32-bit register.

Field descriptions

The HSR bit assignments are:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>IL</td>
<td>ISS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about. Possible values of this field are:

EC == 000000

Unknown reason.
See ISS encoding for exceptions with an unknown reason.

EC == 000001

Trapped WFI or WFE instruction execution.
Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.
See ISS encoding for an exception from a WFI or WFE instruction.

EC == 000011

Trapped MCR or MRC access with (coproc==1111) that is not reported using EC 0b000000.
See ISS encoding for an exception from an MCR or MRC access.

EC == 000100

Trapped MCRR or MRRC access with (coproc==1111) that is not reported using EC 0b000000.
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 000101

Trapped MCR or MRC access with (coproc==1110).
See ISS encoding for an exception from an MCR or MRC access.

EC == 000110

Trapped LDC or STC access.
The only architected uses of these instructions are:
• An STC to write data to memory from DBGDTRRXint.
• An LDC to read data from memory to DBGDTRTXint.
See ISS encoding for an exception from an LDC or STC instruction.
EC == 000111
Access to Advanced SIMD or floating-point functionality trapped by a HCPTR. [TASE, TCP10] control.
Excludes exceptions generated because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.
See ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR.

EC == 001000
Trapped VMRS access, from ID group trap, that is not reported using EC 0b000111.
See ISS encoding for an exception from an MCR or MRC access.

EC == 001100
Trapped MRRC access with (coproc==1110).
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 001110
Illegal exception return to AArch32 state.
See ISS encoding for an exception from an Illegal state or PC alignment fault.

EC == 010001
Exception on SVC instruction execution in AArch32 state routed to EL2.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 010010
HVC instruction execution in AArch32 state, when HVC is not disabled.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 100000
Prefetch Abort from a lower Exception level.
See ISS encoding for an exception from a Prefetch Abort.

EC == 100001
Prefetch Abort taken without a change in Exception level.
See ISS encoding for an exception from a Prefetch Abort.

EC == 100010
PC alignment fault exception.
See ISS encoding for an exception from an Illegal state or PC alignment fault.

EC == 100100
Data Abort from a lower Exception level.
See ISS encoding for an exception from a Data Abort.

EC == 100101
Data Abort taken without a change in Exception level.
See ISS encoding for an exception from a Data Abort.

All other EC values are reserved by ARM, and:

- Unused values in the range 0b0000000 - 0b101100 (0x0 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in "Reserved values in System and memory-mapped registers and translation table entries" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
IL, bit [25]
Instruction length bit. Indicates the size of the instruction that has been trapped to Hyp mode. When this bit is valid, possible values of this bit are:
0  16-bit instruction trapped.
1  32-bit instruction trapped.
This field is RES1 and not valid for the following cases:
• When the EC field is 0b000000, indicating an exception with an unknown reason.
• Prefetch Aborts.
• Data Aborts that do not have valid ISS information, or for which the ISS is not valid.
• When the EC value is 0b001110, indicating an Illegal state exception.

—— Note ———
This is a change from the behavior in ARMv7, where the IL field is UNK/SBZP for the corresponding cases.

The IL field is not valid and is UNKNOWN on an exception from a PC alignment fault.

ISS, bits [24:0]
Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.
The following subsections describe each ISS format.

ISS encoding for exceptions with an unknown reason
This encoding is used by:
• Unknown reason.
The ISS encoding for these exceptions is:

<table>
<thead>
<tr>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RES0</td>
</tr>
</tbody>
</table>

Bits [24:0]
Reserved, RES0.
This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:
• The attempted execution of an instruction bit pattern that has no allocated instruction in the current PE mode, including:
  — A read access using a System register encoding pattern that is not allocated for reads at the current Exception level.
  — A write access using a System register encoding pattern that is not allocated for writes at the current Exception level.
  — Instruction encodings for instructions not implemented in the implementation.
• In Debug state, the attempted execution of an instruction bit pattern that is unallocated in Debug state.
• In Non-debug state, the attempted execution of an instruction bit pattern that is unallocated in Non-debug state.
• The attempted execution of a short vector floating-point instruction.
• In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.

• An exception generated because of the value of one of the SCTL.R.[ITD, SED, CP15BEN] control bits.

• Attempted execution of:
  — An HVC instruction when disabled by HCR.HCD.
  — An HLT instruction when disabled by EDSCR.HDE.

• An exception generated because of the attempted execution of an MSR (banked register) or MRS (banked register) instruction that would access a banked register that is not accessible from the PE mode at which the instruction was executed.

--- Note ---

An exception is generated only if the CONSTRAINED UNPREDICTABLE behavior of the instruction is that it is UNDEFINED, see "MSR/MRS banked registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

---

• Attempted execution, in Debug state, of:
  — A DCPS1 instruction from EL0 when the value of HCR.TGE is 1.
  — A DCPS2 instruction at EL1 or EL0 when the value of EDSCR.HDD is 1 or when EL2 is not implemented.

"Undefined Instruction exception, when HCR.TGE is set to 1" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile describes the configuration settings for a trap that returns an HSR.EC value of 0b000000.

**ISS encoding for an exception from a WFI or WFE instruction**

This encoding is used by:

• Trapped WFI or WFE instruction execution.

  Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

The ISS encoding for these exceptions is:

```
24 23 20 19 1 0
  COND  RES0  
  CV    TI
```

**CV, bit [24]**

Condition code valid. Possible values of this bit are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The COND field is not valid.</td>
</tr>
<tr>
<td>1</td>
<td>The COND field is valid.</td>
</tr>
</tbody>
</table>

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:
- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

**Bits [19:1]**

Reserved, RES0.

**TI, bit [0]**

Trapped instruction. Possible values of this bit are:

- 0: WFI trapped.
- 1: WFE trapped.

"Trapping use of the WFI and WFE instructions" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile describes the configuration settings for this trap.

**ISS encoding for an exception from an MCR or MRC access**

This encoding is used by:

- Trapped MCR or MRC access with (coproc==1111) that is not reported using EC 0b000000.
- Trapped MCR or MRC access with (coproc==1110).
- Trapped VMRS access, from ID group trap, that is not reported using EC 0b000111.

The ISS encoding for these exceptions is:

```
  24 23 20 19 17 16 14 13 10 9 8 5 4 1 0
```

<table>
<thead>
<tr>
<th>24 23</th>
<th>Opc2</th>
<th>Opc1</th>
<th>CRn</th>
<th>10 9</th>
<th>8 5 4 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RES0</td>
</tr>
</tbody>
</table>

**CV, bit [24]**

Condition code valid. Possible values of this bit are:

- 0: The COND field is not valid.
- 1: The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:
- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:
- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

**Opc2, bits [19:17]**

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value 0b000.

**Opc1, bits [16:14]**

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value 0b111.

**CRn, bits [13:10]**

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

**Bit [9]**

Reserved, RES0.

**Rt, bits [8:5]**

The Rt value from the issued instruction, the general-purpose register used for the transfer.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value 0b0000.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:
- 0: Write to System register space. MCR instruction.
- 1: Read from System register space. MRC or VMRS instruction.

The following sections describe configuration settings for traps that are reported using EC value 0b000011:
- "Traps to Hyp mode of Non-secure PL0 and PL1 accesses to the ID registers” in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
• "Traps to Hyp mode of Non-secure PL0 and PL1 accesses to lockdown, DMA, and TCM operations" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure PL1 execution of cache maintenance instructions" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure PL1 execution of TLB maintenance instructions" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure PL1 accesses to the Auxiliary Control Register" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure PL0 and PL1 accesses to Performance Monitors registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure PL1 accesses to the CPACR" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure PL1 accesses to virtual memory control registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "General trapping to Hyp mode of Non-secure PL0 and PL1 accesses to CP15 System registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The following sections describe configuration settings for traps that are reported using EC value 0b000101:

• "ID group 0, Primary device identification registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure CP14 accesses to trace registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Trapping CP14 accesses to Debug ROM registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Trapping CP14 accesses to powerdown debug registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Trapping general CP14 accesses to debug registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The following sections describe configuration settings for traps that are reported using EC value 0b001000:

• "ID group 0, Primary device identification registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "ID group 3, Detailed feature identification registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

**ISS encoding for an exception from an MCRR or MRRC access**

This encoding is used by:

• Trapped MCRR or MRRC access with (coproc==1111) that is not reported using EC 0b000000.

• Trapped MRRC access with (coproc==1110).

The ISS encoding for these exceptions is:
CV, bit [24]

Condition code valid. Possible values of this bit are:

0  The COND field is not valid.
1  The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

COND, bits [23:20]

The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:

• If the instruction is conditional, COND is set to the condition code field value from the instruction.
• If the instruction is unconditional, COND is set to \(0b1110\).

A conditional A32 instruction that is known to pass its condition code check can be presented either:

• With COND set to \(0b1110\), the value for unconditional.
• With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

• CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
• CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to \(0b1110\), or to the value of any condition that applied to the instruction.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

Bits [15:14]

Reserved, RES0.

Rt2, bits [13:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

CRm, bits [4:1]

The CRm value from the issued instruction.
Direction, bit [0]
Indicates the direction of the trapped instruction. The possible values of this bit are:

0 Write to System register space. MCRR instruction.
1 Read from System register space. MRRC instruction.

The following sections describe configuration settings for traps that are reported using EC value 0b000100:

• "Traps to Hyp mode of Non-secure PL1 accesses to virtual memory control registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "General trapping to Hyp mode of Non-secure PL0 and PL1 accesses to CP15 System registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The following sections describe configuration settings for traps that are reported using EC value 0b001100:

• "Traps to Hyp mode of Non-secure CP14 accesses to trace registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Trapping CP14 accesses to Debug ROM registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

ISS encoding for an exception from an LDC or STC instruction
This encoding is used by:

• Trapped LDC or STC access.

The only architected uses of these instructions are:

— An STC to write data to memory from DBGDTRRXint.
— An LDC to read data from memory to DBGDTRTXint.

The ISS encoding for these exceptions is:

```
24 23 20 19 12 11 9 8 5 4 3 1 0
 COND  imm8  RES0  Rn  AM
```

CV, bit [24]
Condition code valid. Possible values of this bit are:

0 The COND field is not valid.
1 The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

COND, bits [23:20]
The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

• If the instruction is conditional, COND is set to the condition code field value from the instruction.
• If the instruction is unconditional, COND is set to 0b1110.
A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

**imm8, bits [19:12]**

The immediate value from the issued instruction.

**Bits [11:9]**

Reserved, RES0.

**Rn, bits [8:5]**

The Rn value from the issued instruction. Valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction.

When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

**Offset, bit [4]**

Indicates whether the offset is added or subtracted:

- 0  Subtract offset.
- 1  Add offset.

This bit corresponds to the U bit in the instruction encoding.

**AM, bits [3:1]**

Addressing mode. The permitted values of this field are:

- 000  Immediate unindexed.
- 001  Immediate post-indexed.
- 010  Immediate offset.
- 011  Immediate pre-indexed.
- 100  Literal unindexed.

LDC instruction in A32 instruction set only.

For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.

- 110  Literal offset.

LDC instruction only.

For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in "Unallocated values in fields of AArch32 System registers and translation table entries" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.
Direction, bit [0]

Indicates the direction of the trapped instruction. The possible values of this bit are:

0  Write to memory. STC instruction.
1  Read from memory. LDC instruction.

"Trapping general CP14 accesses to debug registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile describes the configuration settings for the trap that is reported using EC value 0b000110.

ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR

This encoding is used by:

- Access to Advanced SIMD or floating-point functionality trapped by a HCPTR. {TASE, TCP10} control.
  Excludes exceptions generated because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.

The ISS encoding for these exceptions is:

<table>
<thead>
<tr>
<th>24 23 20 19</th>
<th>6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>RES0</td>
</tr>
<tr>
<td>CV</td>
<td>TA</td>
</tr>
<tr>
<td>coproc</td>
<td>RES0</td>
</tr>
</tbody>
</table>

Excludes exceptions that occur because Advanced SIMD and floating-point functionality is not implemented, or because the value of HCR.TGE or HCR_EL2.TGE is 1. These are reported with EC value 0b000000.

CV, bit [24]

Condition code valid. Possible values of this bit are:

0  The COND field is not valid.
1  The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

Bits [19:6]

Reserved, RES0.

TA, bit [5]

Indicates trapped use of Advanced SIMD functionality. The possible values of this bit are:

0 Exception was not caused by trapped use of Advanced SIMD functionality.
1 Exception was caused by trapped use of Advanced SIMD functionality.

Any use of an Advanced SIMD instruction that is not also a floating-point instruction that is trapped to Hyp mode because of a trap configured in the HCPTR sets this bit to 1.

For a list of these instructions, see "Controls of Advanced SIMD operation that do not apply to floating-point operation" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

Bit [4]

Reserved, RES0.

coproc, bits [3:0]

When the TA field returns the value 1, this field returns the value 1010, otherwise this field is RES0.

The following sections describe the configuration settings for the traps that are reported using EC value 0b000111:

• "General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

ISS encoding for an exception from HVC or SVC instruction execution

This encoding is used by:

• Exception on SVC instruction execution in AArch32 state routed to EL2.
• HVC instruction execution in AArch32 state, when HVC is not disabled.

The ISS encoding for these exceptions is:

<table>
<thead>
<tr>
<th>Bits [24:16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
</tr>
<tr>
<td>RES0</td>
</tr>
</tbody>
</table>

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, this is the value of the imm16 field of the issued instruction.

For an SVC instruction:

• If the instruction is unconditional, then:
  • For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.

- If the instruction is conditional, this field is **UNKNOWN**.

The HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

"Supervisor Call exception, when HCR.TGE is set to 1" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile describes the configuration settings for the trap reported with EC value **0b010001**.

**ISS encoding for an exception from a Prefetch Abort**

This encoding is used by:

- Prefetch Abort from a lower Exception level.
- Prefetch Abort taken without a change in Exception level.

The ISS encoding for these exceptions is:

![ISS encoding diagram]

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>10</td>
<td><strong>FnV</strong>, bit [10]</td>
</tr>
<tr>
<td>9</td>
<td><strong>EA</strong>, bit [9]</td>
</tr>
<tr>
<td>8</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>7</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>6</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>5</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>4</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>3</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>2</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED, RES0</td>
</tr>
<tr>
<td>0</td>
<td>RESERVED, RES0</td>
</tr>
</tbody>
</table>

**FnV, bit [10]**

- FAR not valid, for a Synchronous external abort.
- 0  HIFAR is valid.
- 1  HIFAR is not valid, and holds an **UNKNOWN** value.

This field is only valid if the IFSC code is **010000**. It is RES0 for all other aborts.

**EA, bit [9]**

- External abort type. This bit can provide an **IMPLEMENTATION DEFINED** classification of external aborts.
- For any abort other than an external abort this bit returns a value of 0.

**Bit [8]**

- RESERVED, RES0.

**S1PTW, bit [7]**

- RESERVED, RES0.

**Bit [6]**

- RESERVED, RES0.
IFSC, bits [5:0]

Instruction Fault Status Code. Possible values of this field are:

- 000100 Translation fault
- 001100 Permission fault
- 010000 Synchronous external abort, other than synchronous parity or ECC error
- 011000 Synchronous parity or ECC error on memory access
- 100010 Debug exception, only when the EC value is 0b100001

All other values are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in "Unallocated values in fields of AArch32 System registers and translation table entries" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The following sections describe cases where Prefetch Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100000:

- "Abort exceptions, when HCR.TGE is set to 1" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
- "Routing Debug exceptions to Hyp mode" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

**ISS encoding for an exception from an Illegal state or PC alignment fault**

This encoding is used by:

- Illegal exception return to AArch32 state.
- PC alignment fault exception.

The ISS encoding for these exceptions is:

```
24 0
RES0
```

**Bits [24:0]**

Reserved, RES0.

For more information about the Illegal state exception, see:

- "Illegal changes to PSTATE.M" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
- "Illegal return events from AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
- "Legal exception returns that set PSTATE.IL to 1" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
- "The Illegal Execution state exception" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

For more information about the PC alignment fault exception, see "Branching to an unaligned PC" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
**ISS encoding for an exception from a Data Abort**

This encoding is used by:

- Data Abort from a lower Exception level.
- Data Abort taken without a change in Exception level.

The ISS encoding for these exceptions is:

<table>
<thead>
<tr>
<th>24 23 22 21 20 19</th>
<th>16 15 14 13</th>
<th>11 10 9 8 7 6 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>SRT</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**ISV, bit [24]**

Instruction syndrome valid. Indicates whether the syndrome information in ISS[23:14] is valid.

- 0: No valid instruction syndrome. ISS[23:14] are RES0.

This bit is 0 for all faults except Data Aborts generated by stage 2 address translations for which all the following apply to the instruction that generated the Data Abort exception:

- The instruction is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
- The instruction is not performing register writeback.
- The instruction is not using the PC as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, as described in "Data Aborts in Memory access mode" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

**Note**

In the A32 instruction set, LDR*T and STR*T instructions always perform register writeback and therefore never return a valid instruction syndrome.

**SAS, bits [23:22]**

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

- 00: Byte
- 01: Halfword
- 10: Word
- 11: Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.
SSE, bit [21]
Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:
0  Sign-extension not required.
1  Data item must be sign-extended.
For all other operations this bit is 0.
This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.

Bit [20]
Reserved, RES0.

SRT, bits [19:16]
Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.
This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.

Bit [15]
Reserved, RES0.

AR, bit [14]
Acquire/Release. When ISV is 1, the possible values of this bit are:
0  Instruction did not have acquire/release semantics.
1  Instruction did have acquire/release semantics.
This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.

Bits [13:11]
Reserved, RES0.

FnV, bit [10]
FAR not Valid, for a Synchronous external abort.
0  HDFAR is valid.
1  HDFAR is not valid, and holds an UNKNOWN value.
This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

EA, bit [9]
External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.
For any abort other than an external abort this bit returns a value of 0.

CM, bit [8]
Cache maintenance. For a synchronous fault, identifies fault that comes from a cache maintenance or address translation instruction. For synchronous faults, the possible values of this bit are:
0  Fault not generated by a cache maintenance or address translation instruction.
1  Fault generated by a cache maintenance or address translation instruction.

S1PTW, bit [7]
Reserved, RES0.
WnR, bit [6]
Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction. The possible values of this bit are:

0  Abort caused by a read instruction.
1  Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For an asynchronous Data Abort exception this bit is UNKNOWN.

DFSC, bits [5:0]
Data Fault Status Code. Possible values of this field are:

000100  Translation fault
001100  Permission fault
010000  Synchronous external abort, other than synchronous parity or ECC error
011000  Synchronous parity or ECC error on memory access
010001  SError interrupt
011001  Serror interrupt, parity or ECC error on memory access
100001  Alignment fault
100010  Debug exception, only when the EC value is 0b100100
110100  IMPLEMENTATION DEFINED fault (Cache lockdown fault)
110101  IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault)

All other values are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in "Unallocated values in fields of AArch32 System registers and translation table entries" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The following describe cases where Data Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100100:

• "Abort exceptions, when HCR.TGE is set to 1" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Routing Debug exceptions to Hyp mode" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

The following describe cases that can cause a Data Abort exception that is taken to Hyp mode, and reported in the HSR with EC value of 0b100000 or 0b100100:

• "Hyp mode control of Non-secure access permissions" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

• "Memory fault reporting in Hyp mode" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

Accessing the HSR
This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>
This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c5, c2, 0</td>
<td>100</td>
<td>000</td>
<td>0101</td>
<td>1111</td>
<td>0010</td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>TGE</th>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c5, c2, 0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>RW</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c5, c2, 0</td>
<td>1</td>
<td>-</td>
<td>n/a</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T5==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.10  ID-MMFR0, Memory Model Feature Register 0

The ID-MMFR0 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Configurations**

There are no configuration notes.

**Attributes**

ID-MMFR0 is a 32-bit register.

**Field descriptions**

The ID-MMFR0 bit assignments are:

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-28</td>
<td>InnerShr</td>
</tr>
<tr>
<td>27-24</td>
<td>FCSE</td>
</tr>
<tr>
<td>23-20</td>
<td>AuxReg</td>
</tr>
<tr>
<td>19-16</td>
<td>TCM</td>
</tr>
<tr>
<td>15-12</td>
<td>ShareLvl</td>
</tr>
<tr>
<td>11-8</td>
<td>OuterShr</td>
</tr>
<tr>
<td>7-4</td>
<td>PMSA</td>
</tr>
<tr>
<td>3-0</td>
<td>VMSA</td>
</tr>
</tbody>
</table>

**InnerShr, bits [31:28]**

Innermost Shareability. Indicates the innermost Shareability domain implemented. Defined values are:

- 0000: Implemented as Non-cacheable.
- 0001: Implemented with hardware coherency support.
- 1111: Shareability ignored.

All other values are reserved.

In ARMv8 the permitted values are 0000, 0001, and 1111.

This field is valid only if the implementation supports two levels of Shareability, as indicated by ID-MMFR0.ShareLvl having the value 0001.

When ID-MMFR0.ShareLvl is zero, this field is UNK.

**FCSE, bits [27:24]**

Indicates whether the implementation includes the FCSE. Defined values are:

- 0000: Not supported.
- 0001: Support for FCSE.

All other values are reserved.

In ARMv8 the only permitted value is 0000.

**AuxReg, bits [23:20]**

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

- 0000: None supported.
- 0001: Support for Auxiliary Control Register only.
- 0010: Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

In ARMv8 the only permitted value is 0010.
--- Note ---

Accesses to unimplemented Auxiliary registers are UNDEFINED.

TCM, bits [19:16]

Indicates support for TCMs and associated DMAs. Defined values are:
- 0000 Not supported.
- 0001 Support is IMPLEMENTATION DEFINED. ARMv7 requires this setting.
- 0010 Support for TCM only, ARMv6 implementation.
- 0011 Support for TCM and DMA, ARMv6 implementation.

All other values are reserved.

In ARMv8-R the permitted values are 0000 and 0001.

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of Shareability levels implemented. Defined values are:
- 0000 One level of shareability implemented.
- 0001 Two levels of shareability implemented.

All other values are reserved.

In ARMv8 the only permitted value is 0001.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost Shareability domain implemented. Defined values are:
- 0000 Implemented as Non-cacheable.
- 0001 Implemented with hardware coherency support.
- 1111 Shareability ignored.

All other values are reserved.

In ARMv8 the permitted values are 0000, 0001, and 1111.

PMSA, bits [7:4]

Indicates support for a PMSA. Defined values are:
- 0000 Not supported.
- 0001 Support for IMPLEMENTATION DEFINED PMSA.
- 0010 Support for PMSAv6, with a Cache Type Register implemented.
- 0011 Support for PMSAv7, with support for memory subsections. ARMv7-R profile.
- 0100 Support for ARMv8-R base and limit PMSA.

All other values are reserved.

In ARMv8-R the only permitted value is 0100.

VMSA, bits [3:0]

Indicates support for a VMSA. Defined values are:
- 0000 Not supported.
- 0001 Support for IMPLEMENTATION DEFINED VMSA.
- 0010 Support for VMSAv6, with Cache and TLB Type Registers implemented.
- 0011 Support for VMSAv7, with support for remapping and the Access flag. ARMv7-A profile.
- 0100 As for 0011, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.
0101 As for 0100, and adds support for the Long-descriptor translation table format.
All other values are reserved.
In ARMv8-R the only permitted value is 0000.

Accessing the ID_MMFR0

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c1, 4</td>
<td>000</td>
<td>100</td>
<td>0000</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c1, 4</td>
<td>TGE 0</td>
<td>EL0 RO EL1 RO EL2</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c1, 4</td>
<td>TGE 1</td>
<td>EL0 n/a RO</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TID3==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T0==1, read accesses to this register from EL1 are trapped to Hyp mode.
E2.1.11 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Configurations**

There are no configuration notes.

**Attributes**

ID_MMFR2 is a 32-bit register.

**Field descriptions**

The ID_MMFR2 bit assignments are:

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| HWAccFlg | WFIStall | MemBarr | UniTLB | HvdTLB | L1HvdRng | L1HvdBG | L1HvdFG |
```

**HWAccFlg, bits [31:28]**

Hardware Access Flag. In earlier versions of the ARM Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

- **0000** Not supported.
- **0001** Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

In ARMv8 the only permitted value is **0000**.

**WFIStall, bits [27:24]**

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

- **0000** Not supported.
- **0001** Support for WFI stalling.

All other values are reserved.

In ARMv8 the permitted values are **0000** and **0001**.

**MemBarr, bits [23:20]**

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc==1111) encoding space:

- **0000** None supported.
- **0001** Supported memory barrier System instructions are:
  - Data Synchronization Barrier (DSB).
- **0010** As for 0001, and adds:
  - Instruction Synchronization Barrier (ISB).
  - Data Memory Barrier (DMB).

All other values are reserved.

In ARMv8 the only permitted value is **0010**.
ARM deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

**UniTLB, bits [19:16]**

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation. Defined values are:

- **0000** Not supported.
- **0001** Supported unified TLB maintenance operations are:
  - Invalidate all entries in the TLB.
  - Invalidate TLB entry by VA.
- **0010** As for 0001, and adds:
  - Invalidate TLB entries by ASID match.
- **0011** As for 0010, and adds:
  - Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation.
- **0100** As for 0011, and adds:
  - Invalidate Hyp mode unified TLB entry by VA.
  - Invalidate entire EL1&0 unified TLB.
  - Invalidate entire Hyp mode unified TLB.
- **0101** As for 0100, and adds the following operations: TLBIMVAIL, TLBIMVAILH, TLBIMVAILHIS, TLBIMVAAL, TLBIMVAALH, TLBIMVAALHL. (missing one field)
- **0110** As for 0101, and adds the following operations: TLBIIPAS2IL, TLBIIPAS2LIS, TLBIIPAS2L, TLBIIPAS2S.

All other values are reserved.

In ARMv8-R the only permitted value is **0000**.

**HvdTLB, bits [15:12]**

If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is IMPLEMENTATION DEFINED. ARM deprecates the use of this field by software.

**L1HvdRng, bits [11:8]**

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Defined values are:

- **0000** Not supported.
- **0001** Supported Level 1 Harvard cache maintenance range operations are:
  - Invalidate data cache range by VA.
  - Invalidate instruction cache range by VA.
  - Clean data cache range by VA.
  - Clean and invalidate data cache range by VA.

All other values are reserved.

In ARMv8 the only permitted value is **0000**.

**L1HvdBG, bits [7:4]**

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

- **0000** Not supported.
- **0001** Supported Level 1 Harvard cache background fetch operations are:
  - Fetch instruction cache range by VA.
• Fetch data cache range by VA.

All other values are reserved.

In ARMv8 the only permitted value is 0000.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:

0000  Not supported.
0001  Supported Level 1 Harvard cache foreground fetch operations are:
  • Fetch instruction cache range by VA.
  • Fetch data cache range by VA.

All other values are reserved.

In ARMv8 the only permitted value is 0000.

Accessing the ID_MMFR2

This register can be read using MRC with the following syntax:

MRC <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c1, 6</td>
<td>000</td>
<td>110</td>
<td>0000</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGE</td>
<td>EL0</td>
<td>EL1</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c1, 6</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c1, 6</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

• If HCR.TID3==1, read accesses to this register from EL1 are trapped to Hyp mode.
• If HSTR.T0==1, read accesses to this register from EL1 are trapped to Hyp mode.
E2.1.12 IFSR, Instruction Fault Status Register

The IFSR characteristics are:

**Purpose**
Holds status information about the last instruction fault.

**Configurations**
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**
IFSR is a 32-bit register.

**Field descriptions**
The IFSR bit assignments are:

![Bit assignments diagram]

- **Bits [31:17]**
  - Reserved, RES0.

- **FnV, bit [16]**
  - FAR not Valid, for a Synchronous external abort.
  - 0 IFAR is valid.
  - 1 IFAR is not valid, and holds an UNKNOWN value.
  - This field is only valid for a Synchronous external abort. It is RES0 for all other Prefetch Abort exceptions.

- **Bits [15:13]**
  - Reserved, RES0.

- **ExT, bit [12]**
  - External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.
  - In an implementation that does not provide any classification of external aborts, this bit is RES0.
  - For aborts other than external aborts this bit always returns 0.

- **Bits [11:10]**
  - Reserved, RES0.

- **LPAE, bit [9]**
  - Reserved, RES1.

- **Bits [8:6]**
  - Reserved, RES0.
STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

- 000100: Translation fault
- 001100: Permission fault
- 010000: Synchronous external abort, other than synchronous parity or ECC error
- 011000: Synchronous parity or ECC error on memory access
- 100001: PC alignment fault
- 100010: Debug exception

All other values are reserved.

Accessing the IFSR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c5, c0, l</td>
<td>000</td>
<td>001</td>
<td>0101</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c5, c0, l</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c5, c0, l</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T5==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.1.13 PAR, Physical Address Register

The PAR characteristics are:

Purpose

Returns the output address (OA) from an address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Usage constraints

PAR is accessible as follows:

<table>
<thead>
<tr>
<th></th>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RW</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. Subject to the prioritization rules:

- If HSTR.T7==1, Accesses to this register from EL1 are trapped to Hyp mode.

Configurations

In ARMv8-R, PAR returns a 64-bit value.

RW fields in this register reset to architecturally UNKNOWN values.

Attributes

In ARMv8-R, PAR is a 64-bit register.

Field descriptions

The PAR bit assignments are:

For all register layouts:

F, bit [0]

Indicates whether the instruction performed a successful address translation.

0 Address translation completed successfully.

1 Address translation aborted.

When the instruction returned a 64-bit value to the PAR, PAR.F==0:

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.
On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the translation table descriptors. This applies to the ATTR and SH fields.
- See the NS bit description for constraints on the value it returns.

**ATTR, bits [63:56]**

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR0 and MAIR1.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

**Bits [55:40]**

Reserved, RES0.

**PA, bits [39:12]**

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[39:12].

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

1 Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

**IMP DEF, bit [10]**

IMPLEMENTATION DEFINED.

**NS, bit [9]**

In ARMv8-R, this bit is UNKNOWN.

**SH, bits [8:7]**

Shareability attribute, for the returned output address. Permitted values are:

- 00 Non-shareable.
- 10 Outer Shareable.
- 11 Inner Shareable.

The value 01 is reserved.

--- **Note** ---

This field returns the value 10 for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the translation table descriptor.

**Bits [6:1]**

Reserved, RES0.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

0 Address translation completed successfully.
**When the instruction returned a 64-bit value to the PAR, PAR.F==1:**

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**IMP DEF, bits [63:56]**

*IMPLEMENTATION DEFINED.*

**IMP DEF, bits [55:52]**

*IMPLEMENTATION DEFINED.*

**IMP DEF, bits [51:48]**

*IMPLEMENTATION DEFINED.*

**Bits [47:12]**

Reserved, RES0.

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

1 Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

**Bit [10]**

Reserved, RES0.

**FSTAGE, bit [9]**

Indicates the translation stage at which the translation aborted:

0 Translation aborted because of a fault in the stage 1 translation.

1 Translation aborted because of a fault in the stage 2 translation.

**S2WLK, bit [8]**

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

**Bit [7]**

Reserved, RES0.

**FST, bits [6:1]**

Fault status field. Values are as in the DFSR.STATUS and IFSR.STATUS fields when using the Long-descriptor translation table format.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

1 Address translation aborted.
**Accessing the PAR:**

To access the PAR when accessing as a 64-bit register:

```assembly
MRRC p15,0,<Rt>,<Rt2>,c7 ; Read PAR[31:0] into Rt and PAR[63:32] into Rt2
MCRR p15,0,<Rt>,<Rt2>,c7 ; Write Rt to PAR[31:0] and Rt2 to PAR[63:32]
```

Register access is encoded as follows:

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0000</td>
<td>0111</td>
</tr>
</tbody>
</table>
E2.1.14  **PMCR, Performance Monitors Control Register**

The PMCR characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Configurations**

System register PMCR[6:0] is architecturally mapped to External register PMCR_EL0[6:0].

This register is in the Warm reset domain. Some or all RW fields of this register have defined reset values. On a Warm or Cold reset RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMCR is a 32-bit register.

**Field descriptions**

The PMCR bit assignments are:

<table>
<thead>
<tr>
<th>Bit Assignment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMP, bits [31:24]</td>
<td>Implementer code. This field is RO with an IMPLEMENTATION DEFINED value. The implementer codes are allocated by ARM. Values have the same interpretation as bits [31:24] of the MIDR.</td>
</tr>
<tr>
<td>IDCODE, bits [23:16]</td>
<td>Identification code. This field is RO with an IMPLEMENTATION DEFINED value. Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.</td>
</tr>
<tr>
<td>N, bits [15:11]</td>
<td>Number of event counters. A RO field that indicates the number counters implemented. A value of 0b000000 in this field indicates that only the Cycle Count Register PMCCNTR is implemented. The value of this field is the number of event counters implemented. This value is in the range of 0b000000, in which case only the PMCCNTR is implemented, to 0b111111, which indicates that the PMCCNTR and 31 event counters are implemented. In an implementation that includes EL2, reads of this field from EL1 and EL0 return the value of HDCR.HPMN.</td>
</tr>
<tr>
<td>Bits [10:7]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>LC, bit [6]</td>
<td>Long cycle counter enable. Determines which PMCCNTR bit generates an overflow recorded by PMOVSR[31].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cycle counter overflow on increment that changes PMCCNTR[31] from 1 to 0.</td>
</tr>
<tr>
<td>1</td>
<td>Cycle counter overflow on increment that changes PMCCNTR[63] from 1 to 0.</td>
</tr>
</tbody>
</table>

ARM deprecates use of PMCR.LC = 0.
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**DP, bit [5]**

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

0  PMCCNTR, if enabled, counts when event counting is prohibited.
1  PMCCNTR does not count when event counting is prohibited.

If EL2 is not implemented, this field is RES0, otherwise it is an RW field.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

**X, bit [4]**

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

0  Do not export events.
1  Export events where not prohibited.

This field enables the exporting of events over an event bus to another device, for example to an OPTIONAL trace macrocell. If the implementation does not include such an event bus then this field is RAZ/WI, otherwise it is an RW field.

In an implementation that includes an event bus, no events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

**D, bit [3]**

Clock divider. The possible values of this bit are:

0  When enabled, PMCCNTR counts every clock cycle.
1  When enabled, PMCCNTR counts once every 64 clock cycles.

This bit is RW.

If PMCR.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR.D = 1.

When this register has an architecturally defined reset value, this field resets to 0.

**C, bit [2]**

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

0  No action.
1  Reset PMCCNTR to zero.

This bit is always RAZ.

Resetting PMCCNTR does not clear the PMCCNTR overflow bit to 0.

**P, bit [1]**

Event counter reset. This bit is WO. The effects of writing to this bit are:

0  No action.
1  Reset all event counters accessible in the current EL, not including PMCCNTR, to zero.

This bit is always RAZ.

In EL0 and EL1, if EL2 is implemented, a write of 1 to this bit does not reset event counters that HDCR.HPMN reserves for EL2 use.

In EL2, a write of 1 to this bit resets all the event counters except PMCCNTR.

Resetting the event counters does not clear any overflow bits to 0.
E, bit [0]

Enable. The possible values of this bit are:

0    All counters that are accessible at EL1, including PMCCNTR, are disabled.
1    All counters that are accessible at EL1 are enabled by PMCNTENSET unless the value of HDCR.HPMD is 1, in which case event counting by the following counters is prohibited:
    • Counters in the range of 0-n, where n is one less than the value of HDCR.HPMN.
    • PMCCNTR if the value of PMCR.DP is 1.

This bit is RW.

If EL2 is implemented, this bit does not affect the operation of event counters that HDCR.HPMN reserves for EL2 use.

When this register has an architecturally defined reset value, this field resets to 0.

Accessing the PMCR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c9, c12, 0</td>
<td>000</td>
<td>000</td>
<td>1001</td>
<td>1111</td>
<td>1100</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGE</td>
<td>EL0</td>
<td>EL1</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c9, c12, 0</td>
<td>0</td>
<td>RW</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c9, c12, 0</td>
<td>1</td>
<td>RW</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

Not dependent on other bits:

• If PMUSERENR.EN==0, accesses to this register from EL0 are trapped to Undefined mode.

When EL2 is implemented:

• If HDCR.TPM==1, accesses to this register from EL0 and EL1 are trapped to Hyp mode.
• If HDCR.TPMCR==1, accesses to this register from EL0 and EL1 are trapped to Hyp mode.
• If HSTR.T9==1, accesses to this register from EL0 and EL1 are trapped to Hyp mode.
E2.1.15  **SCTLR, System Control Register**

The SCTLR characteristics are:

**Purpose**

Provides the top-level control of the system, including its memory system.

**Configurations**

Fields in this register reset to architecturally unknown values.

**Attributes**

SCTLR is a 32-bit register.

**Field descriptions**

The SCTLR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>TE, bit [30] T32 Exception Enable. This bit controls whether exceptions to an Exception Level that is executing at EL1 are taken to A32 or T32 state:</td>
</tr>
<tr>
<td>29-28</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>27-26</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>25</td>
<td>EE, bit [25] The value of the PSTATE.E bit on branch to an exception vector or coming out of reset. The possible values of this bit are:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset.</td>
</tr>
<tr>
<td>1</td>
<td>Big-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset.</td>
</tr>
</tbody>
</table>

If an implementation does not provide big-endian support for data accesses at Exception levels higher than EL0, this bit is RES0.
If an implementation does not provide little-endian support for data accesses at Exception levels higher than EL0, this bit is RES1.

**Bit [24]**

Reserved, RES0.

**Bits [23:22]**

Reserved, RES1.

**FI, bit [21]**

Fast Interrupts enable. This bit is a read-only copy of the HSCTLR.FI bit.

**UWXN, bit [20]**

Unprivileged write permission implies EL1 XN (Execute-never). This bit can force all memory regions that are writable at EL0 to be treated as XN for accesses from software executing at EL1. The possible values of this bit are:

0  This control has no effect on memory access permissions.
1  Any region that is writable at EL0 forced to XN for accesses from software executing at EL1.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN. The possible values of this bit are:

0  This control has no effect on memory access permissions.
1  Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

**nTWE, bit [18]**

Traps EL0 execution of WFE instructions to Undefined mode.

0  Any attempt to execute a WFE instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
1  This control has no effect on the EL0 execution of WFE instructions.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

--- **Note** ---

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

---

**BR, bit [17]**

Background Region enable. When the EL1-controlled MPU is enabled, this bit controls how an EL1 access that does not map to any MPU memory region is handled:

0  EL1-controlled MPU Background region disabled. Any EL1 transaction that does not match an EL1-controlled MPU region results in a fault.
1  EL1-controlled MPU Background region enabled. For EL1 transactions that do not match an EL1-controlled MPU region, the EL1 Background region attributes are used and access is permitted subject to any stage 2 checks.

This bit only applies to EL1 accesses. An EL0 access that does not match an EL1-controlled MPU region always results in a Translation fault.
nTWI, bit [16]

Traps EL0 execution of WFI instructions to Undefined mode.

0  Any attempt to execute a WFI instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

1  This control has no effect on the EL0 execution of WFI instructions.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

Bits [15:13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL1 and EL0:

0  All instruction access to Normal memory from EL1 and EL0 are Non-cacheable for all levels of instruction and unified cache.
   If the value of SCTLR.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

1  All instruction access to Normal memory from EL1 and EL0 can be cached at all levels of instruction and unified cache.
   If the value of SCTLR.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

Instruction accesses to Normal memory from EL1 and EL0 are Cacheable regardless of the value of the SCTLR.I bit if the value of HCR.DC is 1.

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL0 and EL1.

0  SETEND instruction execution is enabled at EL0 and EL1.

1  SETEND instructions are UNDEFINED at EL0 and EL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL1 and EL0.

0  All IT instruction functionality is enabled at EL1 and EL0.

1  Any attempt at EL1 or EL0 to execute any of the following is UNDEFINED:
   •  All encodings of the IT instruction with hw1[3:0]!=1000.
• All encodings of the subsequent instruction with the following values for hw1:
  11xxxxxxxxxxxxx
  All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
  1011xxxxxxxxxxxx
  All instructions in "Miscellaneous 16-bit instructions" in the ARM®
  Architecture Reference Manual, ARMv8, for ARMv8-A architecture
  profile.
  10100xxxxxxxxxxx
  ADD Rd, PC, #imm
  01001xxxxxxxxxxx
  LDR Rd, [PC, #imm]
  0100x1xx1111xxx
  ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
  010001xx1xxxx111
  ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers
  UNPREDICTABLE cases with BLX Rn.
  These instructions are always UNDEFINED, regardless of whether they would pass or fail
  the condition code check that applies to them as a result of being in an IT block.
  It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:
  • A 16-bit instruction, that can only be followed by another 16-bit instruction.
  • The first half of a 32-bit instruction.
  This means that, for the situations that are UNDEFINED, either the second 16-bit
  instruction or the 32-bit instruction is UNDEFINED.
  An implementation might vary dynamically as to whether IT is treated as a 16-bit
  instruction or the first half of a 32-bit instruction.

ITD is optional. If it is not implemented then this bit is RAZ/WI.

UNK, bit [6]
Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

CP15BEN, bit [5]
System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System
instructions in the (coproc==1111) encoding space from EL1 and EL0:
  0 EL0 and EL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is
    UNDEFINED.
  1 EL0 and EL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is
    enabled.

CP15BEN is optional. If it is not implemented then this bit is RAO/WI.

Bits [4:3]
Reserved, RES1.

C, bit [2]
Cacheability control, for data accesses at EL1 and EL0:
  0 All data access to Normal memory from EL1 and EL0 are Non-cacheable for all levels
    of data and unified cache.
  1 All data access to Normal memory from EL1 and EL0 can be cached at all levels of data
    and unified cache.

The PE ignores SCLTR.C and data accesses to Normal memory from EL1 and EL0 are Cacheable
if the value of HCR.DC is 1.
A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0:

0 Alignment fault checking disabled when executing at EL1 or EL0.
Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element or data elements being accessed.

1 Alignment fault checking enabled when executing at EL1 or EL0.
All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element or data elements 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.

M, bit [0]

MPU enable for EL1-controlled MPU. Possible values of this bit are:

0 EL1-controlled MPU disabled.
See the SCTLR.I field for the behavior of instruction accesses to Normal memory.

1 EL1-controlled MPU enabled.
The PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if the value of HCR.TGE is 1.

Accessing the SCTLR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c1, c0, 0</td>
<td>000</td>
<td>000</td>
<td>0001</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c1, c0, 0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c1, c0, 0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
When EL2 is implemented:

- If HCR.TVM == 1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM == 1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T1 == 1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2 New System registers

This section contains the description of the System registers that are new in ARMv8-R from ARMv8-A.
E2.2.1 HMPUIR, Hypervisor MPU Type Register

The HMPUIR characteristics are:

**Purpose**

Identifies the number of regions supported by the EL2-controlled MPU.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

HMPUIR is a 32-bit register.

**Field descriptions**

The HMPUIR bit assignments are:

<table>
<thead>
<tr>
<th>Bit assignments</th>
<th>Bit assignment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>RESERVED, RES0</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>REGION</td>
<td>The number of EL2-controlled MPU regions implemented.</td>
</tr>
</tbody>
</table>

An EL2-controlled MPU region controls EL2 access and stage 2 of EL1 and EL0 access.

**Accessing the HMPUIR**

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c0, c0, 4</td>
<td>100</td>
<td>100</td>
<td>0000</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c0, c0, 4</td>
<td>TGE</td>
<td>EL0 EL1 EL2</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c0, c0, 4</td>
<td>0</td>
<td>- - RW</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c0, c0, 4</td>
<td>1</td>
<td>- n/a RW</td>
</tr>
</tbody>
</table>
Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

• If HSTR.T0==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.2  **HPRBAR, Hypervisor Protection Region Base Address Register**

The HPRBAR characteristics are:

**Purpose**

Provides indirect access to the base address of the EL2-controlled MPU region currently defined by HPRSELR.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

HPRBAR is a 32-bit register.

**Field descriptions**

The HPRBAR bit assignments are:

```
31 6 5 4 3 2 1 0
  |   |   |   |   |   |
  |   |   |   |   |   |
  BASE

BASE, bits [31:6]
Address[31:6] concatenated with zeroes to form Address[31:0], the lower inclusive limit used as the base address for the selected memory region.
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bit [5]
Reserved, RES0.

SH[1:0], bits [4:3]
Shareability attribute:
00 Non-shareable.
01 Reserved, CONSTRAINED UNPREDICTABLE.
10 Outer Shareable.
11 Inner Shareable.
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

AP[2:1], bits [2:1]
Access permissions attribute:
00 Read/write at EL2, no access at EL1 or EL0.
01 Read/write at EL2, EL1 and EL0.
10 Read-only at EL2, no access at EL1 or EL0.
11 Read-only at EL2, EL1 and EL0.
```
For accesses at EL1 or EL0, the access permission attribute applies only when the value of HCR.VM is 1, in which case these stage 2 permissions are combined with the stage 1 permissions for the access.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

XN, bit [0]

Execute never.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

Accessing the HPRBAR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c3, 0</td>
<td>100</td>
<td>000</td>
<td>0110</td>
<td>1111</td>
<td>0011</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c3, 0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c3, 0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T6==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.3 HPRBAR\(<n>\), Hypervisor Protection Region Base Address Registers, \(n = 0 - 31\)

The HPRBAR\(<n>\) characteristics are:

**Purpose**

Provides access to the base addresses for the first 32 defined EL2-controlled MPU regions.

**Configurations**

These registers are available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

HPRBAR\(<n>\) is a 32-bit register.

**Field descriptions**

The HPRBAR\(<n>\) bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>BASE</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>XN</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AP[2:1]</td>
</tr>
<tr>
<td>2</td>
<td>SH[1:0]</td>
</tr>
<tr>
<td>1</td>
<td>RES0</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**BASE, bits \([31:6]\)**

Address\([31:6]\) concatenated with zeroes to form Address\([31:0]\), the lower inclusive limit used as the base address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [5]**

Reserved, RES0.

**SH[1:0], bits \([4:3]\)**

Shareability attribute:

- 00 Non-shareable.
- 01 Reserved, CONSTRAINED UNPREDICTABLE.
- 10 Outer Shareable.
- 11 Inner Shareable.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**AP[2:1], bits \([2:1]\)**

Access permissions attribute:

- 00 Read/write at EL2, no access at EL1 or EL0.
- 01 Read/write at EL2, EL1 and EL0.
- 10 Read-only at EL2, no access at EL1 or EL0.
- 11 Read-only at EL2, EL1 and EL0.
For accesses at EL1 or EL0, the access permission attribute applies only when the value of HCR.VM is 1, in which case these stage 2 permissions are combined with the stage 1 permissions for the access.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**XN, bit [0]**

Execute never.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

### Accessing the HPRBAR<n>

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>0110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Direct access is provided to HPRBAR0-HPRBAR31. HPRBAR<n>, can be accessed via MCR/MRC p15,i,<Rt>,c6,ck,j,k where:

i = '00' : n[4]

j = '1' : n[3:1]

k = n[0] : '00'

Registers beyond the number of implemented regions are UNALLOCATED.

### Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

### Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

• If HSTR.T10==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.4 HPRENR, Hypervisor Protection Region Enable Register

The HPRENR characteristics are:

**Purpose**

Provides direct access to the HPRLAR.EN bits for EL2-controlled MPU regions 0 to 31.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

HPRENR is a 32-bit register.

**Field descriptions**

The HPRENR bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 0</td>
<td>ENABLE_bits</td>
</tr>
</tbody>
</table>

**ENABLE_bits, bits [31:0]**

An alias of the HPRLAR[31:0].EN bits.

Bits associated with unimplemented regions are RAZ/WI.

When this register has an architecturally defined reset value, this field resets to 0.

**Accessing the HPRENR**

This register can be read using MRC with the following syntax:

```
MRC <syntax>
```

This register can be written using MCR with the following syntax:

```
MCR <syntax>
```

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c1, l</td>
<td>100</td>
<td>001</td>
<td>0110</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c1, l</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c1, l</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T6==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.5 HPRLAR, Hypervisor Protection Region Limit Address Register

The HPRLAR characteristics are:

**Purpose**

Provides indirect access to the limit address of the EL2-controlled MPU region currently defined by HPRSEL.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

The AttrIndx[2:0] field is associated with the HMAIR0 and HMAIR1 registers.

**Attributes**

HPRLAR is a 32-bit register.

**Field descriptions**

The HPRLAR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>LIMIT</td>
</tr>
<tr>
<td>6</td>
<td>EN</td>
</tr>
<tr>
<td>5</td>
<td>AttrIndx[2:0]</td>
</tr>
<tr>
<td>4</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**LIMIT, bits [31:6]**

Address[31:6] concatenated with the value 0x3F to form Address[31:0], the upper inclusive limit address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bits [5:4]**

Reserved, RES0.

**AttrIndx[2:0], bits [3:1]**

Selects attributes from within the associated Hyp Memory Attribute Indirection Register:

- 000 Select the Attr0 field from HMAIR0.
- 001 Select the Attr1 field from HMAIR0.
- 010 Select the Attr2 field from HMAIR0.
- 011 Select the Attr3 field from HMAIR0.
- 100 Select the Attr4 field from HMAIR1.
- 101 Select the Attr5 field from HMAIR1.
- 110 Select the Attr6 field from HMAIR1.
- 111 Select the Attr7 field from HMAIR1.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.
EN, bit [0]

Region enable.
0 Region disabled.
1 Region enabled.

When this register has an architecturally defined reset value, this field resets to 0.

Accessing the HPRLAR

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c3, 1</td>
<td>100</td>
<td>001</td>
<td>0110</td>
<td>1111</td>
<td>0011</td>
</tr>
</tbody>
</table>

Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c3, 1</td>
<td>0</td>
<td>-/a RW</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c3, 1</td>
<td>1</td>
<td>n/a RW</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

• If HSTR.T6==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.6  HPRLAR<n>, Hypervisor Protection Region Limit Address Registers, n = 0 - 31

The HPRLAR<n> characteristics are:

**Purpose**

Provides access to the limit addresses for the first 32 defined EL2-controlled MPU regions.

**Configurations**

These registers are available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).
The AttrIndx[2:0] field is associated with the HMAIR0 and HMAIR1 registers.

**Attributes**

HPRLAR<n> is a 32-bit register.

**Field descriptions**

The HPRLAR<n> bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>LIMIT</td>
</tr>
<tr>
<td>6-5</td>
<td></td>
</tr>
<tr>
<td>4-3</td>
<td></td>
</tr>
<tr>
<td>2-1</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>0</td>
<td>EN</td>
</tr>
</tbody>
</table>

**LIMIT, bits [31:6]**

Address[31:6] concatenated with the value 0x3F to form Address[31:0], the upper inclusive limit address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bits [5:4]**

Reserved, RES0.

**AttrIndx[2:0], bits [3:1]**

Selects attributes from within the associated Hyp Memory Attribute Indirection Register:

- 000 Select the Attr0 field from HMAIR0.
- 001 Select the Attr1 field from HMAIR0.
- 010 Select the Attr2 field from HMAIR0.
- 011 Select the Attr3 field from HMAIR0.
- 100 Select the Attr4 field from HMAIR1.
- 101 Select the Attr5 field from HMAIR1.
- 110 Select the Attr6 field from HMAIR1.
- 111 Select the Attr7 field from HMAIR1.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EN, bit [0]**

Region enable.

0 Region disabled.
Region enabled. When this register has an architecturally defined reset value, this field resets to 0.

**Accessing the HPRLAR<n>**

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>0110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGE EL0 EL1 EL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Direct access is provided to HPRLAR0-HPRLAR31. HPRLAR<n>, can be accessed via MCR/MRC p15,i,<Rt>,c6, ej,k where:

- i = ‘00’ : n[4]
- j = ‘1’ : n[3:1]
- k = n[0] : ‘00’

Registers beyond the number of implemented regions are UNALLOCATED.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T10==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.7 HPRSELR, Hypervisor Protection Region Selector Register

The HPRSELR characteristics are:

**Purpose**

Selects the region number for the EL2-controlled MPU region associated with the HPRBAR and HPRLAR registers.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

HPRSELR is a 32-bit register.

**Field descriptions**

The HPRSELR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>7-0</td>
<td>REGION, region number HPRSELR&lt;N:0&gt;,</td>
</tr>
<tr>
<td></td>
<td>where the value is zero extended if N &lt; 7.</td>
</tr>
<tr>
<td></td>
<td>The size of the region field, N,</td>
</tr>
<tr>
<td></td>
<td>is $\log_2$(Number of regions supported), rounded up to an integer.</td>
</tr>
<tr>
<td></td>
<td>For X implemented regions, memory</td>
</tr>
<tr>
<td></td>
<td>region numbering starts at 0 and</td>
</tr>
<tr>
<td></td>
<td>increments by 1 to the value X-1.</td>
</tr>
<tr>
<td></td>
<td>Writing a value greater than or equal to X is UNPREDICTABLE.</td>
</tr>
<tr>
<td></td>
<td>When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.</td>
</tr>
</tbody>
</table>

**Accessing the HPRSELR**

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c6, c2, 1</td>
<td>100</td>
<td>001</td>
<td>0110</td>
<td>1111</td>
<td>0010</td>
</tr>
</tbody>
</table>
Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T6=1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.8 MPUIR, MPU Type register

The MPUIR characteristics are:

**Purpose**

Identifies the number of regions supported by the EL1-controlled MPU.

Regions only support unified instruction and data address spaces; ARMv7 supported separate instruction and data regions controlled by bit 0.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

MPUIR is a 32-bit register.

**Field descriptions**

The MPUIR bit assignments are:

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>15-8</td>
<td>REGION, bits [15:8]</td>
</tr>
<tr>
<td>7-0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**REGION, bits [15:8]**

The number of EL1-controlled MPU regions implemented.

An EL1-controlled MPU region controls EL1 and EL0 access.

**Bits [7:0]**

Reserved, RES0.

**Accessing the MPUIR**

This register can be read using MRC with the following syntax:

```
MRC <syntax>
```

This register can be written using MCR with the following syntax:

```
MCR <syntax>
```

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c0, c0, 4</td>
<td>000</td>
<td>100</td>
<td>0000</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>
Accessibility

The register is accessible in software as follows:

```
<syntax>
Control   Accessibility
TGE  EL0  EL1  EL2
p15, 0, <Rt>, c0, c0, 4  0  -  RW  RW
p15, 0, <Rt>, c0, c0, 4  1  -  n/a  RW
```

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T0==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TID1==1, read accesses to this register from EL1 are trapped to Hyp mode.
E2.2.9 PRBAR, Protection Region Base Address Register

The PRBAR characteristics are:

**Purpose**

Provides indirect access to the base address of the EL1-controlled MPU region currently defined by PRSEL.R.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

PRBAR is a 32-bit register.

**Field descriptions**

The PRBAR bit assignments are:

<table>
<thead>
<tr>
<th>Bit 31-6</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BASE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AP[2:1]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SH[1:0]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RES0</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BASE, bits [31:6]**

Address[31:6] concatenated with zeroes to form Address[31:0], the lower inclusive limit used as the base address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [5]**

Reserved, RES0.

**SH[1:0], bits [4:3]**

Shareability attribute:

- 00 Non-shareable.
- 01 Reserved, CONSTRAINED UNPREDICTABLE.
- 10 Outer Shareable.
- 11 Inner Shareable.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**AP[2:1], bits [2:1]**

Access permissions attribute:

- 00 Read/write at EL1, no access at EL0.
- 01 Read/write, at EL1 or EL0.
- 10 Read-only at EL1, no access at EL0.
- 11 Read-only at EL1 and EL0.
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

XN, bit [0]

Execute never.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Accessing the PRBAR**

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c3, 0</td>
<td>000</td>
<td>000</td>
<td>0110</td>
<td>1111</td>
<td>0011</td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>TGE</th>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c3, 0</td>
<td>0</td>
<td>RW</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c3, 0</td>
<td>1</td>
<td>n/a</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T6==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.10 PRBAR<\(n\)>, Protection Region Base Address Registers, \(n = 0 - 31\)

The PRBAR<\(n\)> characteristics are:

**Purpose**

Provides access to the base addresses for the first 32 defined EL1-controlled MPU regions.

**Configurations**

These registers are available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

PRBAR<\(n\)> is a 32-bit register.

**Field descriptions**

The PRBAR<\(n\)> bit assignments are:

### BASE, bits [31:6]

Address[31:6] concatenated with zeroes to form Address[31:0], the lower inclusive limit used as the base address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [5]**

Reserved, RES0.

**SH[1:0], bits [4:3]**

Shareability attribute:

- 00  Non-shareable.
- 01  Reserved, CONSTRAINED UNPREDICTABLE.
- 10  Outer Shareable.
- 11  Inner Shareable.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**AP[2:1], bits [2:1]**

Access permissions attribute:

- 00  Read/write at EL1, no access at EL0.
- 01  Read/write, at EL1 or EL0.
- 10  Read-only at EL1, no access at EL0.
- 11  Read-only at EL1 and EL0.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.
XN, bit [0]

Execute never.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

### Accessing the PRBAR<n>

This register can be read using MRC with the following syntax:

**MRC <syntax>**

This register can be written using MCR with the following syntax:

**MCR <syntax>**

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>0110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>TGE</td>
<td>EL0 EL1 EL2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>- RW RW</td>
</tr>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>1</td>
<td>- n/a RW</td>
</tr>
</tbody>
</table>

Direct access is provided to PRBAR0-PRBAR31. PRBAR<n>, can be accessed via MCR/MRC p15,i,<(Rt),c6,cj,k

where:

i = '00' : n[4]

j = '1' : n[3:1]

k = n[0] : '00'

Registers beyond the number of implemented regions are UNALLOCATED.

### Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T10==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.11  PRLAR, Protection Region Limit Address Register

The PRLAR characteristics are:

**Purpose**

Provides indirect access to the limit address of the EL1-controlled MPU region currently defined by PRSEL.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

The AttrIndx[2:0] field is associated with the MAIR0 and MAIR1 registers.

**Attributes**

PRLAR is a 32-bit register.

**Field descriptions**

The PRLAR bit assignments are:

```
+----------------+-----------------+----------------+-----------------+----------------+----------------+
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |
+----------------+-----------------+----------------+-----------------+----------------+----------------+
| LIMIT, bits [31:6] | Address[31:6] concatenated with the value 0x3F to form Address[31:0], the upper inclusive limit address for the selected memory region. | When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN. |
| AttrIndx[2:0], bits [3:1] | Selects attributes from within the associated Memory Attribute Indirection Register: |
| 000 | Select the Attr0 field from MAIR0. |
| 001 | Select the Attr1 field from MAIR0. |
| 010 | Select the Attr2 field from MAIR0. |
| 011 | Select the Attr3 field from MAIR0. |
| 100 | Select the Attr4 field from MAIR1. |
| 101 | Select the Attr5 field from MAIR1. |
| 110 | Select the Attr6 field from MAIR1. |
| 111 | Select the Attr7 field from MAIR1. |
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN. |
```

LIMIT, bits [31:6]

Address[31:6] concatenated with the value 0x3F to form Address[31:0], the upper inclusive limit address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

Bits [5:4]

Reserved, RES0.

AttrIndx[2:0], bits [3:1]

Selects attributes from within the associated Memory Attribute Indirection Register:

000  Select the Attr0 field from MAIR0.
001  Select the Attr1 field from MAIR0.
010  Select the Attr2 field from MAIR0.
011  Select the Attr3 field from MAIR0.
100  Select the Attr4 field from MAIR1.
101  Select the Attr5 field from MAIR1.
110  Select the Attr6 field from MAIR1.
111  Select the Attr7 field from MAIR1.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.
EN, bit [0]
Region enable:
0  Region disabled.
1  Region enabled.
When this register has an architecturally defined reset value, this field resets to 0.

Accessing the PRLAR
This register can be read using MRC with the following syntax:
MRC <syntax>
This register can be written using MCR with the following syntax:
MCR <syntax>
This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c3, l</td>
<td>000</td>
<td>001</td>
<td>0110</td>
<td>1111</td>
<td>0011</td>
</tr>
</tbody>
</table>

Accessibility
The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c3, l</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c3, l</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Traps and Enables
For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:
- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T6==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.12 PRLAR<\(n\)>, Protection Region Limit Address Registers, \(n = 0 - 31\)

The PRLAR<\(n\)> characteristics are:

**Purpose**

Provides access to the limit addresses for the first 32 defined EL1-controlled MPU regions.

**Configurations**

These registers are available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

The AttrIndx[2:0] field is associated with the MAIR0 and MAIR1 registers.

**Attributes**

PRLAR<\(n\)> is a 32-bit register.

**Field descriptions**

The PRLAR<\(n\)> bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-6</td>
<td>Limit, bits [31:6]</td>
</tr>
<tr>
<td>5-4</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>3-1</td>
<td>AttrIndx[2:0], bits [3:1]</td>
</tr>
<tr>
<td>0</td>
<td>EN, bit [0]</td>
</tr>
</tbody>
</table>

**Limit, bits [31:6]**

Address[31:6] concatenated with the value 0x3F to form Address[31:0], the upper inclusive limit address for the selected memory region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bits [5:4]**

Reserved, RES0.

**AttrIndx[2:0], bits [3:1]**

Selects attributes from within the associated Memory Attribute Indirection Register:

- 000 Select the Attr0 field from MAIR0.
- 001 Select the Attr1 field from MAIR0.
- 010 Select the Attr2 field from MAIR0.
- 011 Select the Attr3 field from MAIR0.
- 100 Select the Attr4 field from MAIR1.
- 101 Select the Attr5 field from MAIR1.
- 110 Select the Attr6 field from MAIR1.
- 111 Select the Attr7 field from MAIR1.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EN, bit [0]**

Region enable:

- 0 Region disabled.
Region enabled.

When this register has an architecturally defined reset value, this field resets to 0.

**Accessing the PRLAR<\(n\)>**

This register can be read using MRC with the following syntax:

\[
\text{MRC \langle syntax\rangle}
\]

This register can be written using MCR with the following syntax:

\[
\text{MCR \langle syntax\rangle}
\]

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>0110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>TGE EL0 EL1 EL2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>RW RW</td>
</tr>
<tr>
<td>p15, &lt;opc1&gt;, &lt;Rt&gt;, c6, &lt;CRm&gt;, &lt;opc2&gt;</td>
<td>1</td>
<td>- n/a RW</td>
</tr>
</tbody>
</table>

Direct access is provided to PRLAR0-PRLAR31. PRLAR<\(n\)> can be accessed via MCR/MRC \(p15,i,<Rt>,c6,cj,k\) where:

- \(i = ‘00’ \): \(n[4]\)
- \(j = ‘1’ \): \(n[3:1]\)
- \(k = n[0] \): ‘00’

Registers beyond the number of implemented regions are UNALLOCATED.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T10==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.13 PRSELR, Protection Region Selector Register

The PRSELR characteristics are:

**Purpose**

Selects the region number for the EL1-controlled MPU region associated with the PRBAR and PRLAR registers.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

PRSELR is a 32-bit register.

**Field descriptions**

The PRSELR bit assignments are:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30-24</th>
<th>Bit 23-16</th>
<th>Bit 15-8</th>
<th>Bit 7-0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>REGION</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RES0.

REGION, bits [7:0]

The region number, PRSELR<7:0>, where the value is zero extended if N < 7.

The size of the region field, N, is \( \log_2(\text{Number of regions supported}) \), rounded up to an integer.

For X implemented regions, memory region numbering starts at 0 and increments by 1 to the value X-1.

Writing a value greater than or equal to X is UNPREDICTABLE.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Accessing the PRSELR**

This register can be read using MRC with the following syntax:

MRC <syntax>

This register can be written using MCR with the following syntax:

MCR <syntax>

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c2, 1</td>
<td>000</td>
<td>001</td>
<td>0110</td>
<td>1111</td>
<td>0010</td>
</tr>
</tbody>
</table>
Accessibility

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c2, 1</td>
<td>0</td>
<td>RW RW</td>
</tr>
<tr>
<td>p15, 0, &lt;Rt&gt;, c6, c2, 1</td>
<td>1</td>
<td>n/a RW</td>
</tr>
</tbody>
</table>

Traps and Enables

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HCR.TVM==1, write accesses to this register from EL1 are trapped to Hyp mode.
- If HCR.TRVM==1, read accesses to this register from EL1 are trapped to Hyp mode.
- If HSTR.T6==1, accesses to this register from EL1 are trapped to Hyp mode.
E2.2.14 VSCTLR, Virtualization System Control register

The VSCTLR characteristics are:

**Purpose**

Provides control and configuration information for PMSA virtualization.

**Configurations**

This register is available in all implementations that support the ARMv8 Protected Management System Architecture (PMSA).

**Attributes**

VSCTLR is a 32-bit register.

**Field descriptions**

The VSCTLR bit assignments are:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-24</td>
<td>RES0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>23-16</td>
<td>VMID</td>
<td>Virtual machine identifier.</td>
</tr>
<tr>
<td>15-3</td>
<td>RES0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>2</td>
<td>S2NIE</td>
<td>Stage 2 Normal Interrupt Enable.</td>
</tr>
<tr>
<td>1</td>
<td>S2DMAD</td>
<td>Stage 2 Device Multiword Access Disable.</td>
</tr>
</tbody>
</table>

**Bits [31:24]**

Reserved, RES0.

**VMID, bits [23:16]**

Virtual machine identifier. When this register has an architecturally defined reset value, this field resets to 0.

**Bits [15:3]**

Reserved, RES0.

**S2NIE, bit [2]**

Stage 2 Normal Interrupt Enable.

0    Feature disabled.
1    Feature enabled. Multi-word accesses are interruptible, regardless of stage 1 attributes, where the access:
   • Uses base-restore addressing.
   • Is executed at EL1 or EL0.
   • Is an access to memory marked as Normal in the corresponding EL2-controlled MPU region.

In ARMv8-R, this field only has any effect when HSCTLR.FI == 1.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**S2DMAD, bit [1]**

Stage 2 Device Multiword Access Disable.

0    Feature disabled.
Feature enabled. Multi-word accesses at EL1 or EL0 that span an aligned 64-bit boundary generate a stage 2 Permission Fault if the memory region is marked as Device in the corresponding EL2-controlled MPU region.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [0]**

Reserved, RES0.

**Accessing the VSCTLR**

This register can be read using MRC with the following syntax:

`MRC <syntax>`

This register can be written using MCR with the following syntax:

`MCR <syntax>`

This syntax is encoded with the following settings in the instruction encoding:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>opc1</th>
<th>opc2</th>
<th>CRn</th>
<th>coproc</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p15, 4, &lt;Rt&gt;, c2, c0, 0</td>
<td>100</td>
<td>000</td>
<td>0010</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Accessibility**

The register is accessible in software as follows:

<table>
<thead>
<tr>
<th>&lt;syntax&gt;</th>
<th>Control</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TGE</td>
<td>EL0</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c2, c0, 0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>p15, 4, &lt;Rt&gt;, c2, c0, 0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see "Synchronous exception prioritization for exceptions taken to AArch32 state" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

When EL2 is implemented:

- If HSTR.T2==1, accesses to this register from EL1 are trapped to Hyp mode.
Part F

Differences in ARMv8-R Debug from ARMv8-A
Chapter F1
Differences in ARMv8-R Debug from ARMv8-A

This chapter describes the Debug features of the ARMv8-R AArch32 profile. It contains the following sections:

- Differences from ARMv8-A invasive debug on page F1-186.
- Differences from ARMv8-A non-invasive debug on page F1-187.
- Differences from ARMv8-A external debug on page F1-188.
Because ARMv8-R implements only AArch32 state:

- Self-hosted debug is always disabled at EL2.
- Self-hosted debug is always enabled at EL1 and EL0.
- EL2 can trap any attempted use of self-hosted debug to Hyp mode.
- The Software Step exception is not implemented.
F1.2 Differences from ARMv8-A non-invasive debug

When the value of \texttt{HDCR.HPMD} is 1, event counting by the following counters is prohibited at EL2:

- Counters in the range 0 - n, where n is one less than the value of \texttt{HDCR.HPMN}.
- If the value of \texttt{PMCR.DP} is 1, \texttt{PMCCNTR}.

Other counters are controlled by \texttt{HDCR.HPME}. See \texttt{AArch32.CountEvents()}.

If \texttt{ExternalHypNoninvasiveDebugEnabled()} returns FALSE, then the following apply:

- When the value of \texttt{HDCR.EPMAD} is 1, an external debugger cannot access counters in the range \(a-b\), including the values \(a\) and \(b\), where:
  - \(a\) is the value of \texttt{HDCR.HPMN}.
  - \(b\) is one less than the value of \texttt{PMCR.N}.

For a counter \(x\) that is in this range, for accesses using the external debug interface:

- \texttt{PMEVTYPER<\(n\)>} and \texttt{PMEVCNTR<\(n\)>} are RAZ/WI.
- The register bits \texttt{PMOVSCLR_EL0}, \texttt{PMOVSSET}, \texttt{PMCNTENSET}, \texttt{PMCNTENCLR}, \texttt{PMINTENSET}, and \texttt{PMINTENCLR} are RAZ/WI.
- A write of 1 to \texttt{PMCR.P} does not reset \texttt{PMEVCNTR<\(n\)>} to 0.

- When the value of \texttt{HDCR.EPMAD} is 1, a read of \texttt{PMCFGR.N} using the external debug interface returns the value of \texttt{HDCR.HPMN}.

- PC sample-based profiling and Trace are prohibited both:
  - At EL2.
  - When the value of \texttt{HCR.TGE} is 1.

If \texttt{ExternalNoninvasiveDebugEnabled()} returns FALSE, the external debugger cannot access the PMU registers.

See also \texttt{AllowExternalPMUAccess()} and \texttt{ExternalHypNoninvasiveDebugEnabled()}. 


F1.3 Differences from ARMv8-A external debug

This section describes how the authentication interface might prevent an external debugger from debugging software executing at

- EL2
- EL0 when the value of HCR.TGE is 1.

See also HaltingAllowed().

F1.3.1 Required debug authentication

Any implementation must provide the debug authentication to control:

- Whether the PE can halt.
- Whether non-invasive debug is permitted.

The pseudocode functions, together with the following conditions, define the architectural requirements for debug authentication.

- If ExternalInvasiveDebugEnabled() returns FALSE, then ExternalHypInvasiveDebugEnabled() returns FALSE.
- If ExternalNoninvasiveDebugEnabled() returns FALSE, then ExternalHypNoninvasiveDebugEnabled() returns FALSE.
- If ExternalInvasiveDebugEnabled() returns TRUE, then ExternalNoninvasiveDebugEnabled() returns TRUE.
- If ExternalHypInvasiveDebugEnabled() returns TRUE, then ExternalHypNoninvasiveDebugEnabled() returns TRUE.

F1.3.2 Recommended authentication interface

The details of the debug authentication interface are IMPLEMENTATION DEFINED but ARM recommends the use of the CoreSight interface, which includes the following signals for external debug authentication:

- DBGEN.
- HIDEN.
- NIDEN.
- HNIDEN.

shared/debug/authentication/Debug_authentication defines the authentication signals DBGEN, HIDEN, NIDEN, and HNIDEN. See the ARM® CoreSight™ v2.0 Architecture Specification.

If EL2 is not implemented, HIDEN and HNIDEN are not implemented and the PE behaves as if these signals were tied LOW.

It is IMPLEMENTATION DEFINED how the authentication signals are driven. The architecture permits, but does not require, PEs within a cluster to have independent authentication interfaces. ARM recommends that any Trace extension has the same authentication interface as the PE it is connected to. For more information, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4.

Table F1-1 shows the debug authentication pseudocode functions and the recommended implementations.

<table>
<thead>
<tr>
<th>Pseudocode function</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExternalHypNoninvasiveDebugEnabled()</td>
<td>(DBGEN OR NIDEN) AND (HIDEN OR HNIDEN)</td>
</tr>
<tr>
<td>ExternalHypInvasiveDebugEnabled()</td>
<td>(DBGEN AND HIDEN)</td>
</tr>
<tr>
<td>ExternalNoninvasiveDebugEnabled()</td>
<td>(DBGEN OR NIDEN)</td>
</tr>
<tr>
<td>ExternalInvasiveDebugEnabled()</td>
<td>DBGEN</td>
</tr>
</tbody>
</table>
F1.3.3 Halting enabled and prohibited

When `ExternalHypInvasiveDebugEnabled()` returns FALSE:

- Halting is prohibited:
  - From EL2.
  - When the value of `HCR.TGE` is 1.
- Writes to `EDRCR.CBRRQ` are ignored.
- Writes to `EDPRCR.CWRR` are ignored.
- `EDSCR.TDA` is ignored at EL2.
- `EDSCR.INTdis` does not mask interrupts that are taken to EL2.

For information on behavioral restrictions in ARMv8-R when moving between [EL2 when `ExternalHypInvasiveDebugEnabled()` returns FALSE (halting is prohibited)] and [EL1 or EL0 when `ExternalHypInvasiveDebugEnabled()` returns TRUE and the value of `HCR.TGE` is 0 (halting is allowed)], see the information in ARMv8-A about moving between [Secure state with `ExternalSecureInvasiveDebugEnabled()` returns FALSE (halting is prohibited)] and [Non-secure state with `ExternalInvasiveDebugEnabled()` returns TRUE (halting is allowed)].

When the PE is in Non-debug state, `EDSCR.HDD` returns the inverse of the value that `ExternalHypInvasiveDebugEnabled()` returns.

F1.3.4 Behavior in Debug state

On entry to Debug state:

- If the PE is at EL2 or the value of `HCR.TGE` is 1, `EDSCR.HDD` is set to 0.
- Otherwise, `EDSCR.HDD` is set to the inverse of `ExternalHypInvasiveDebugEnabled()`.

When the PE is in Debug state, the value of `EDSCR.HDD` cannot change.

When the value of `EDSCR.HDD` is 1 in Debug state:

- `DCPS2` is UNDEFINED.
- Instructions that are executed at EL1 or EL0 that are configured by an EL2 control register to trap to EL2 become UNDEFINED.
- Faults that are generated by the EL2 MPU that would be taken as an exception to EL2 are taken as an exception to EL1. The EL1 fault syndrome registers are updated as for any other exception that is taken to EL1 but the fault status in `DFSR.STATUS` or `DFSR.FS[4:0]` is set to Debug event.

F1.3.5 Halting Step Debug events

The following criteria determine the value that is written to `EDESR.SS` on taking an exception during stepping an instruction:

- Whether halting is allowed at the Exception level that the exception targets. If halting is allowed, the PE must step into the exception.
  - If halting is prohibited, it must step over it.
- If stepping over the exception, whether the exception handler will return to re-execute the instruction or return to the next instruction. That is, whether the preferred return address of the exception is the instruction itself or the next instruction.

This means that the behavior in the active-not-pending state is modified for ARMv8-R.

PE behavior in the active-not-pending state if an exception or debug event is generated

The PE sets `EDESR.SS` according to all of the following:

- The type of exception.
- The target Exception level of the exception.
Depending on the result of `ExternalHypNoninvasiveDebugEnabled()`, whether halting is prohibited at EL2.

If an exception or debug event is generated, the PE sets `EDESR.SS` to 1 if one of the following applies:

- A synchronous exception is generated by the instruction and at least one of the following applies:
  - The exception is taken to EL1.
  - The exception is taken to EL2 and `ExternalHypInvasiveDebugEnabled()` returns TRUE.
  - The exception is an HVC exception.

- An asynchronous exception is generated before executing an instruction, and either:
  - The exception is taken to EL1.
  - The exception is taken to EL2 and `ExternalHypInvasiveDebugEnabled()` returns TRUE.

Otherwise `EDESR.SS` is unchanged. These other cases are either:

- No instruction is executed because either:
  - An asynchronous exception is taken to EL2 and `ExternalHypInvasiveDebugEnabled()` returns FALSE.
  - An asynchronous debug event caused entry to Debug state.

- An instruction is executed and either:
  - `ExternalHypInvasiveDebugEnabled()` returns FALSE and the instruction generates a synchronous exception, other than an HVC exception, that is taken to EL2.
  - The instruction generates a synchronous debug event and causes entry to Debug state.

For more information about halting step, see section Halting Step debug events in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

### Table F1-2 Summary of active-not-pending state behavior

<table>
<thead>
<tr>
<th>Event</th>
<th>Target Exception level</th>
<th><code>ExternalHypInvasiveDebugEnabled()</code></th>
<th>Value written to <code>EDESR.SS</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>No exception or debug event</td>
<td>Not applicable</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>HVC exception</td>
<td>EL2</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>Reset</td>
<td>Highest</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>Exception, other than HVC exception</td>
<td>EL1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EL2</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Debug event</td>
<td>Debug state</td>
<td>x</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

**F1.3.6 Access to debug registers**

`HIDEN` has no effect on the access permissions for debug registers. See `AllowExternalDebugAccess()`.

Software must be aware that if `HIDEN` is LOW and `DBGEN` is HIGH, an external debugger can modify breakpoint and watchpoint control registers. Self-hosted debug is disabled at EL2, so an untrusted debugger cannot impact software executing at EL2. However, it can impact software executing at EL0 when the value of `HCR.TGE` is 1. If `DBGEN` is HIGH, then such a debugger can also halt the processor at EL1 or EL0 when the value of `HCR.TGE` is 0.

Writes to `OSLAR_EL1` by an external debugger return an error when `ExternalInvasiveDebugEnabled()` returns FALSE.
Part G

ARMv8-R External Debug Registers
Chapter G1
ARMv8-R External Debug Registers

This chapter contains the description of the external debug registers that are new or redefined in ARMv8-R from ARMv8-A. It contains the following section:

•  *ARMv8-R external debug register list* on page G1-194.
G1.1 ARMv8-R external debug register list

The external debug interface register map is described by:

- External debug interface register map.
- Performance Monitors external register views on page G1-195.

All other locations are reserved. For information on the external debug registers that are unchanged with respect to ARMv8-A, see the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

Table G1-1 External debug interface register map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x020</td>
<td>EDESR</td>
</tr>
<tr>
<td>0x024</td>
<td>EDECR</td>
</tr>
<tr>
<td>0x030</td>
<td>EDWAR[31:0]</td>
</tr>
<tr>
<td>0x034</td>
<td>EDWAR[63:32]</td>
</tr>
<tr>
<td>0x080</td>
<td>DBGDTRRX_EL0</td>
</tr>
<tr>
<td>0x084</td>
<td>EDITR</td>
</tr>
<tr>
<td>0x088</td>
<td>EDSCR</td>
</tr>
<tr>
<td>0x08C</td>
<td>DGDTRTX_EL0</td>
</tr>
<tr>
<td>0x090</td>
<td>EDRCR</td>
</tr>
<tr>
<td>0x094</td>
<td>EDACR</td>
</tr>
<tr>
<td>0x098</td>
<td>EDECCR</td>
</tr>
<tr>
<td>0x0A0</td>
<td>EDPCSR</td>
</tr>
<tr>
<td>0x0A4</td>
<td>EDCIDSR</td>
</tr>
<tr>
<td>0x0A8</td>
<td>EDVIDSR</td>
</tr>
<tr>
<td>0x300</td>
<td>OSLAR_EL1</td>
</tr>
<tr>
<td>0x310</td>
<td>EDPRCR</td>
</tr>
<tr>
<td>0x314</td>
<td>EDPRSR</td>
</tr>
<tr>
<td>0x400+16×n</td>
<td>DBGBVR&lt;n&gt;_EL1[31:0]</td>
</tr>
<tr>
<td>0x404+16×n</td>
<td>DBGBVR&lt;n&gt;_EL1[63:32]</td>
</tr>
<tr>
<td>0x408+16×n</td>
<td>DBGBCR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>0x800+16×n</td>
<td>DBGWVR&lt;n&gt;_EL1[31:0]</td>
</tr>
<tr>
<td>0x804+16×n</td>
<td>DBGWVR&lt;n&gt;_EL1[63:32]</td>
</tr>
<tr>
<td>0x808+16×n</td>
<td>DBGWCR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>0xC00–0xCFc</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0x000</td>
<td>MIDR_EL1</td>
</tr>
<tr>
<td>0x020</td>
<td>EDPFR[31:0]</td>
</tr>
<tr>
<td>0x024</td>
<td>EDPFR[63:32]</td>
</tr>
</tbody>
</table>
### Table G1-1 External debug interface register map (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x028</td>
<td>EDDFR[31:0]</td>
</tr>
<tr>
<td>0x02C</td>
<td>EDDFR[63:32]</td>
</tr>
<tr>
<td>0xD30</td>
<td>RAZ</td>
</tr>
<tr>
<td>0xD38</td>
<td>RAZ</td>
</tr>
<tr>
<td>0x60</td>
<td>EDAA32PFR[31:0]</td>
</tr>
<tr>
<td>0x64</td>
<td>EDAA32PFR[63:32]</td>
</tr>
<tr>
<td>0xE80-0xEC</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0xF00-0xFFC</td>
<td>Management registers, see Table G1-3 on page G1-196</td>
</tr>
</tbody>
</table>

### Table G1-2 Performance Monitors external register views

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>0x0F8</td>
<td>PMCCNTR_EL0[31:0]</td>
</tr>
<tr>
<td>0xFC</td>
<td>PMCCNTR_EL0[63:32]</td>
</tr>
<tr>
<td>0x400</td>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>0x47C</td>
<td>PMCCFILTR_EL0</td>
</tr>
<tr>
<td>0x600-0xEC</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0xA00-0xBFC</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0xC00</td>
<td>PMCNTENSET_EL0</td>
</tr>
<tr>
<td>0xC20</td>
<td>PMCNTENCLR_EL0</td>
</tr>
<tr>
<td>0xC40</td>
<td>PMINTENSET_EL1</td>
</tr>
<tr>
<td>0xC60</td>
<td>PMINTENCLR_EL1</td>
</tr>
<tr>
<td>0xC80</td>
<td>PMOVSCLR_EL0</td>
</tr>
<tr>
<td>0xCA0</td>
<td>PMSWINC_EL0</td>
</tr>
<tr>
<td>0xCC0</td>
<td>PMOVSET_EL0</td>
</tr>
<tr>
<td>0xD80-0xDFC</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0xE00</td>
<td>PMCFGR</td>
</tr>
<tr>
<td>0xE04</td>
<td>PMCR_EL0</td>
</tr>
<tr>
<td>0xE20</td>
<td>PMCEID0</td>
</tr>
<tr>
<td>0xE24</td>
<td>PMCEID1</td>
</tr>
<tr>
<td>0xE80-0xEC</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0xF00-0xFFC</td>
<td>Management registers, see Table G1-3 on page G1-196</td>
</tr>
</tbody>
</table>
Table G1-3 shows the external management register maps for the following registers:

| ED | These are the external debug registers. |
| CTI | These are the Cross-trigger interface registers. |
| PMU | These are the Performance Monitors registers. |

Some of these registers are required for compliance with the ARMv8 architecture and some are required for compliance with the CoreSight architecture. Other registers are optional. See the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* for more information.

### Table G1-3 CoreSight interface register map of management registers

<table>
<thead>
<tr>
<th>Offset</th>
<th>Mnemonic</th>
<th>ED</th>
<th>CTI</th>
<th>PMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF00</td>
<td>EDITCTRL</td>
<td>CTITCTRL</td>
<td>PMITCTRL</td>
<td></td>
</tr>
<tr>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
<td>CTICLAIMSET</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>CTICLAIMCLR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0xFA8</td>
<td>EDDEVAFF0</td>
<td>CTIDEVAFF0</td>
<td>PMDEVAFF0</td>
<td></td>
</tr>
<tr>
<td>0xFAC</td>
<td>EDDEVAFF1</td>
<td>CTIDEVAFF1</td>
<td>PMDEVAFF1</td>
<td></td>
</tr>
<tr>
<td>0xFB0</td>
<td>EDLAR</td>
<td>CTILAR</td>
<td>PMLAR</td>
<td></td>
</tr>
<tr>
<td>0xFB4</td>
<td>EDLSR</td>
<td>CTILSR</td>
<td>PMSLR</td>
<td></td>
</tr>
<tr>
<td>0xFB8</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>CTIAUTHSTATUS</td>
<td>PMAUTHSTATUS</td>
<td></td>
</tr>
<tr>
<td>0xFBC</td>
<td>EDDEVARCH</td>
<td>CTIDEVARCH</td>
<td>PMDEVARCH</td>
<td></td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVARCH2</td>
<td>CTIDEVARCH2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0xFC4</td>
<td>EDDEVID1</td>
<td>CTIDEVID1</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0xFC8</td>
<td>EDDEVID</td>
<td>CTIDEVID</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0xFCC</td>
<td>EDDEVTYPE</td>
<td>CTIDEVTYPE</td>
<td>PMDEVTYPE</td>
<td></td>
</tr>
</tbody>
</table>
Chapter G2
Description of the Redefined External Debug Registers

This chapter contains the description of the external debug registers that are redefined in ARMv8-R from ARMv8-A. It contains the following sections:

- Redefined external debug registers on page G2-198.
G2.1 Redefined external debug registers

This section contains the description of the external debug registers that are redefined in ARMv8-R from ARMv8-A.
G2.1.1   DBGAUTHSTATUS_EL1, Debug Authentication Status register

The DBGAUTHSTATUS_EL1 characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Usage constraints**

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Default</th>
<th>RO</th>
</tr>
</thead>
</table>

**Configurations**

External register DBGAUTHSTATUS_EL1 is architecturally mapped to System register DBGAUTHSTATUS.

DBGAUTHSTATUS_EL1 is in the Debug power domain.

**Attributes**

DBGAUTHSTATUS_EL1 is a 32-bit register.

**Field descriptions**

The DBGAUTHSTATUS_EL1 bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>HNID</td>
<td>HID</td>
<td>SNID</td>
<td>SID</td>
<td>NSID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:12]**

Reserved, RES0.

**HNID, bits [11:10]**

Hyp non-invasive debug. Possible values of this field are:

- **00** Separate Hyp enable not implemented, or EL2 not implemented.
- **10** Implemented and disabled. ExternalHypNoninvasiveDebugEnabled() == FALSE.
- **11** Implemented and enabled. ExternalHypNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.

For ARMv8-R:

- If EL2 is implemented, HNID[1] is RAO.
- If EL2 is not implemented, HNID[1:0] are RAZ.

**HID, bits [9:8]**

Hyp invasive debug. Possible values of this field are:

- **00** Separate Hyp enable not implemented, or EL2 not implemented.
- **10** Implemented and disabled. ExternalHypInvasiveDebugEnabled() == FALSE.
- **11** Implemented and enabled. ExternalHypInvasiveDebugEnabled() == TRUE.
Other values are reserved.

For ARMv8-R:
- If EL2 is implemented, HID[1] is RAO.
- If EL2 is not implemented, HID[1:0] are RAZ.

SNID, bits [7:6]
Secure non-invasive debug. Possible values of this field are:
00 Not implemented.
Other values are reserved.

SID, bits [5:4]
Secure invasive debug. Possible values of this field are:
00 Not implemented.
Other values are reserved.

NSNID, bits [3:2]
Non-secure non-invasive debug. Possible values of this field are:
10 Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
11 Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.
Other values are reserved.

NSID, bits [1:0]
Non-secure invasive debug. Possible values of this field are:
10 Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
11 Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.
Other values are reserved.

Accessing the DBGAUTHSTATUS_EL1:

DBGAUTHSTATUS_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFB8</td>
</tr>
</tbody>
</table>
G2.1.2 EDAA32PFR, External Debug AArch32 Processor Feature Register

The EDAA32PFR characteristics are:

**Purpose**
Provides information about implemented PE features.

**Usage constraints**
This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>RO</td>
</tr>
</tbody>
</table>

**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              | 16              | 15              | 12              | 11              | 8               | 7               | 4               |
| RES0            | EL3             | EL2             | PMSA            | VMSA            |                 |                 |                 |
+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
--- Note ---
EDPFR. [EL1, EL0] indicate whether EL1 and EL0 can only be executed in AArch32 state.

PMSA, bits [7:4]
Indicates support for a PMSA. Defined values are:
0000 PMSA not supported.
0100 Support for an ARMv8-R PMSA_v8-32.
All other values are reserved. In ARMv8-R, the only permitted value is 0100.

VMSA, bits [3:0]
Indicates support for a VMSA. When the PMSA field is nonzero, determines support for a VMSA.
When the PMSA field is 0000, VMSA is supported. Defined values are:
0000 VMSA not supported.
All other values are reserved. In ARMv8-R, the only permitted value is 0000.

Accessing the EDAA32PFR:
EDAA32PFR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x060</td>
</tr>
</tbody>
</table>
G2.1.3 EDDEVARCH, External Debug Device Architecture register

The EDDEVARCH characteristics are:

**Purpose**

Identifies the programmers’ model architecture of the external debug component.

**Usage constraints**

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Default</th>
<th>RO</th>
</tr>
</thead>
</table>

**Configurations**

EDDEVARCH is in the Debug power domain.

Implementation of this register is OPTIONAL.

**Attributes**

EDDEVARCH is a 32-bit register.

**Field descriptions**

The EDDEVARCH bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>21 20 19</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECT</td>
<td>REVISION</td>
<td>ARCHID</td>
<td></td>
</tr>
</tbody>
</table>

**PRESENT**

When set to 1, indicates that the DEVARCH is present.

This field is 1 in ARMv8-R.

**ARCHITECT, bits [31:21]**

Defines the architecture of the component. For debug, this is ARM Limited.

Bits [31:28] are the JEP 106 continuation code, 0x4.

Bits [27:21] are the JEP 106 ID code, 0x3B.

**PRESENT, bit [20]**

When set to 1, indicates that the DEVARCH is present.

This field is 1 in ARMv8-R.

**REVISION, bits [19:16]**

Defines the architecture revision. For architectures defined by ARM this is the minor revision.

For debug, the revision defined by ARMv8-R is 0x0.

All other values are reserved.

**ARCHID, bits [15:0]**

Defines this part to be an ARMv8-R debug component. For architectures defined by ARM this is further subdivided.

For debug:

- Bits [15:12] are the architecture version, 0x6.
- Bits [11:0] are the architecture part number, 0xA05.
This corresponds to the ARMv8-R debug architecture version.

**Accessing the EDDEVARCH:**

EDDEVARCH can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFFFBC</td>
</tr>
</tbody>
</table>
G2.1.4 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

Purpose

Provides top-level information about the debug system.

Usage constraints

This register is accessible as follows:

<table>
<thead>
<tr>
<th></th>
<th>Off</th>
<th>DLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDDFR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>RO</td>
</tr>
</tbody>
</table>

Configurations

It is IMPLEMENTATION DEFINED whether EDDFR is implemented in the Core power domain or in the Debug power domain.

Attributes

EDDFR is a 64-bit register.

Field descriptions

The EDDFR bit assignments are:

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>RES0</td>
</tr>
<tr>
<td>[31:28]</td>
<td>CTX_CMPs</td>
</tr>
<tr>
<td>[27:24]</td>
<td>RES0</td>
</tr>
<tr>
<td>[23:20]</td>
<td>WRP</td>
</tr>
<tr>
<td>[19:16]</td>
<td>RES0</td>
</tr>
<tr>
<td>[15:12]</td>
<td>BRPs</td>
</tr>
<tr>
<td>[11:8]</td>
<td>PMUVer</td>
</tr>
<tr>
<td>[7:4]</td>
<td>TraceVer</td>
</tr>
<tr>
<td>[3:0]</td>
<td>RES0</td>
</tr>
</tbody>
</table>

Bits [63:32]

Reserved, RES0.

CTX_CMPs, bits [31:28]

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1. The value of 0b0000 is reserved.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1. The value of 0b0000 is reserved.

PMUVer, bits [11:8]

Performance Monitors Extension version. Indicates whether System register interface to Performance Monitors Extension is implemented. Defined values are:

- 0000: Performance Monitors Extension System registers not implemented.
- 0001: Performance Monitors Extension System registers implemented, PMUv3.
1111  IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported.

All other values are reserved.

**TraceVer, bits [7:4]**

Trace support. Indicates whether System register interface to a trace macrocell is implemented. Defined values are:

- `0000` Trace macrocell System registers not implemented.
- `0001` Trace macrocell System registers implemented.

All other values are reserved.

A value of `0b0000` only indicates that no System register interface to a trace macrocell is implemented. A trace macrocell might nevertheless be implemented without a System register interface.

**Bits [3:0]**

Reserved, RES0.

### Accessing the EDDFR:

EDDFR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xD28</td>
</tr>
</tbody>
</table>

EDDFR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xD2C</td>
</tr>
</tbody>
</table>
G2.1.5  EDPCSR, External Debug Program Counter Sample Register

The EDPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Usage constraints

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>SLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
</tbody>
</table>

Configurations

EDPCSR is in the Core power domain. RW fields in this register reset to architecturally UNKNOWN values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External Debug reset.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented.

Attributes

EDPCSR is a 32-bit register.

Field descriptions

The EDPCSR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>PC Sample word, EDPCSR</td>
</tr>
</tbody>
</table>

Bits [31:0]

PC Sample word, EDPCSR. Bits [31:0] of the sampled instruction address value. The translation regime that EDPCSR samples is determined by EDVIDSR.E2.

For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the access has no side effects. Otherwise, Reading EDPCSR has the side-effect of updating EDCIDSR and EDVIDSR. However:

- If the PE is in Debug state, or PC Sample-based Profiling is prohibited, EDPCSR reads as 0xFFFFFFFF and EDCIDSR and EDVIDSR become UNKNOWN.
- If the PE is in Reset state, the sampled value is unknown and EDCIDSR and EDVIDSR become UNKNOWN.
- If no instruction has been retired since the PE left Reset state, Debug state, or a state where PC Sample-based Profiling is prohibited, the sampled value is UNKNOWN and EDCIDSR and EDVIDSR become UNKNOWN.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.
Accessing the EDPCSR:

EDPCSR[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xA0</td>
</tr>
</tbody>
</table>
G2.1.6 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

**Purpose**

Provides information about implemented PE features.

**Usage constraints**

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Configurations**

It is IMPLEMENTATION DEFINED whether EDPFR is implemented in the Core power domain or in the Debug power domain.

For general information about the interpretation of the ID registers see "Principles of the ID scheme for fields in ID registers" in the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

**Attributes**

EDPFR is a 64-bit register.

**Field descriptions**

The EDPFR bit assignments are:

<table>
<thead>
<tr>
<th>Bits [63:28]</th>
<th>63</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>GIC</td>
<td>AdvSIMD</td>
<td>FP</td>
<td>EL3</td>
<td>EL2</td>
<td>EL1</td>
<td>EL0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GIC, bits [27:24]**

System register GIC interface support. Defined values are:

- 0000  No System register interface to the GIC is supported.
- 0001  System register interface to version 3.0 of the GIC CPU interface is supported.

All other values are reserved.

**AdvSIMD, bits [23:20]**

Advanced SIMD. Defined values are:

- 0000  Advanced SIMD is implemented.
- 1111  Advanced SIMD is not implemented.

All other values are reserved.

**FP, bits [19:16]**

Floating-point. Defined values are:

- 0000  Floating-point is implemented.
- 1111  Floating-point is not implemented.
All other values are reserved.

**EL3, bits [15:12]**

AArch64 EL3 Exception level handling. Defined values are:

- **0000** EL3 is not implemented or cannot be executed in AArch64 state.
- **0001** EL3 can be executed in AArch64 state only.
- **0010** EL3 can be executed in either AArch64 or AArch32 state.

When the value of EDAA32PFR.EL3 is non-zero, this field must be 0000.

All other values are reserved.

In ARMv8-R the only permitted value is 0000.

**EL2, bits [11:8]**

AArch64 EL2 Exception level handling. Defined values are:

- **0000** EL2 is not implemented or cannot be executed in AArch64 state.
- **0001** EL2 can be executed in AArch64 state only.
- **0010** EL2 can be executed in either AArch64 or AArch32 state.

When the value of EDAA32PFR.EL2 is non-zero, this field must be 0000.

All other values are reserved.

In ARMv8-R the only permitted value is 0000.

**EL1, bits [7:4]**

AArch64 EL1 Exception level handling. Defined values are:

- **0000** EL1 can be executed in AArch32 state only.
- **0001** EL1 can be executed in AArch64 state only.
- **0010** EL1 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

In ARMv8-R the only permitted value is 0000.

**EL0, bits [3:0]**

AArch64 EL0 Exception level handling. Defined values are:

- **0000** EL0 can be executed in AArch32 state only.
- **0001** EL0 can be executed in AArch64 state only.
- **0010** EL0 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

In ARMv8-R the only permitted value is 0000.

**Accessing the EDPFR:**

EDPFR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x020</td>
</tr>
</tbody>
</table>

EDPFR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x024</td>
</tr>
</tbody>
</table>
G2.1.7  EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

Purpose

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

Usage constraints

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>SLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>WI</td>
<td>WO</td>
</tr>
</tbody>
</table>

Configurations

EDRCR is in the Core power domain.

Attributes

EDRCR is a 32-bit register.

Field descriptions

The EDRCR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>30</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>29</td>
<td>CSE</td>
</tr>
<tr>
<td>28</td>
<td>CSPA</td>
</tr>
<tr>
<td>27</td>
<td>CBRRQ</td>
</tr>
</tbody>
</table>

Bits [31:5]

Reserved, RES0.

CBRRQ, bit [4]

Allow imprecise entry to Debug state. The actions on writing to this bit are:

0   No action.
1   Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

Writes to this bit are also ignored if either:

- External debugging is not enabled (ExternalInvasiveDebugEnabled() == FALSE).
- EL2 is implemented and Hyp mode external debugging is not enabled (ExternalHypInvasiveDebugEnabled() == FALSE).

CSPA, bit [3]

Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:

0   No action.
1 Clear the EDSCR.PipeAdv bit to 0.

CSE, bit [2]
Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:

0 No action.
1 Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.

Bits [1:0]
Reserved, RES0.

Accessing the EDRCR:
EDCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x090</td>
</tr>
</tbody>
</table>
G2.1.8 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Usage constraints

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>SLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RW</td>
</tr>
</tbody>
</table>

Configurations

EDSCR is in the Core power domain. Some or all RW fields of this register have defined reset values. These apply only on a Cold reset. The register is not affected by a Warm reset and is not affected by an External Debug reset.

Attributes

EDSCR is a 32-bit register.

Field descriptions

The EDSCR bit assignments are:

Bit [31]
Reserved, RES0.

RXfull, bit [30]
DTRRX full. This bit is RO.

When this register has an architecturally defined reset value, this field resets to 0.
TXfull, bit [29]

DTRTX full. This bit is RO.
When this register has an architecturally defined reset value, this field resets to 0.

ITO, bit [28]

ITR overrun. This bit is RO.
If the PE is in Non-debug state, this bit is UNKNOWN. ITO is set to 0 on entry to Debug state.

RXO, bit [27]

DTRRX overrun. This bit is RO.
When this register has an architecturally defined reset value, this field resets to 0.

TXU, bit [26]

DTRTX underrun. This bit is RO.
When this register has an architecturally defined reset value, this field resets to 0.

PipeAdv, bit [25]

Pipeline advance. This bit is RO. Set to 1 every time the PE pipeline retires one or more instructions.
Cleared to 0 by a write to EDRCR.CSPA.
The architecture does not define precisely when this bit is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing.

ITE, bit [24]

ITR empty. This bit is RO.
If the PE is in Non-debug state, this bit is UNKNOWN. It is always valid in Debug state.

INTdis, bits [23:22]

Interrupt disable. Disables taking interrupts (including virtual interrupts, asynchronous Data Abort exceptions, and SError interrupts) in Non-Debug state.

If ExternalInvasiveDebugEnabled() = FALSE, the value of this field is ignored.
If ExternalInvasiveDebugEnabled() = TRUE, the possible values of this field are:

00  Do not disable interrupts.
01  Disable interrupts taken to EL1.
10  Reserved.
11  Disable interrupts taken to EL1. If ExternalHypInvasiveDebugEnabled() == TRUE, also disable all other interrupts.

The value of INTdis does not affect whether an interrupt is a WFI wake-up event, but can mask an interrupt as a WFE wake-up event.
If EL2 is not implemented, the value 0b01 is reserved. If programmed with a reserved value the PE behaves as if INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.
When this register has an architecturally defined reset value, this field resets to 0.

TDA, bit [21]

Traps accesses to DBGBCR<\n>, DBGBVR<\n>, DBGBXVR<\n>, DBGWCR<\n>, and DBGWVR<\n>.
The possible values of this field are:

0  Accesses to the registers listed above do not generate a Software Access debug event.
1  Accesses to the registers listed above generate a Software Access debug event, if OSLSR.OSLK is 0 and if halting is allowed.

When this register has an architecturally defined reset value, this field resets to 0.
MA, bit [20]
Memory access mode. Controls use of memory-access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:
0 Normal access mode.
1 Memory access mode.

Bit [19]
Reserved, RES0.

Bit [18]
Reserved, RES1.

Bit [17]
Reserved, RES0.

Bit [16]
Reserved, RES1.

HDD, bit [15]
Hyp debug disabled. This bit is RO. On entry to Debug state:
• If entering Debug state in Hyp mode, this bit is set to 0.
• Otherwise, this bit is set to the inverse of ExternalHypInvasiveDebugEnabled().

In Debug state, the value of the HDD bit does not change, even if ExternalHypInvasiveDebugEnabled() changes.

In Non-debug state, HDD returns the inverse of ExternalHypInvasiveDebugEnabled(). If the authentication signals that control ExternalHypInvasiveDebugEnabled() change, a context synchronization operation is required to guarantee their effect.

If EL2 is not implemented, this bit is RES0.

HDE, bit [14]
Halting debug enable. The possible values of this field are:
0 Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
1 Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

When this register has an architecturally defined reset value, this field resets to 0.

Bits [13:10]
Reserved, RES0.

EL, bits [9:8]
Exception level. Read-only. In Debug state, this gives the current EL of the PE.
In Non-debug state, this field is RAZ.

A, bit [7]
Asynchronous Data Abort or SError interrupt pending. Read-only. In Debug state, indicates whether an asynchronous Data Abort or SError interrupt is pending:
• If HCR.{AMO, TGE} = {1, 0} and in EL0 or EL1, a virtual asynchronous Data Abort or SError interrupt.
• Otherwise, a physical asynchronous Data Abort or SError interrupt.
0 No asynchronous Data Abort or SError interrupt pending.
1 Asynchronous Data Abort or SError interrupt pending.
A debugger can read EDSCR to check whether an asynchronous Data Abort or SError interrupt is pending without having to execute further instructions. A pending asynchronous Data Abort or SError might indicate data from target memory is corrupted.

UNKNOWN in Non-debug state.

**ERR, bit [6]**
Cumulative error flag. This field is RO. It is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When this register has an architecturally defined reset value, this field resets to 0.

**STATUS, bits [5:0]**
Debug status flags. This field is RO.
The possible values of this field are:

- 000010 PE is in Non-debug state.
- 000001 PE is restarting, exiting Debug state.
- 000111 Breakpoint.
- 010011 External debug request.
- 011011 Halting step, normal.
- 011111 Halting step, exclusive.
- 100011 OS Unlock Catch.
- 100111 Reset Catch.
- 101011 Watchpoint.
- 101111 HLT instruction.
- 110011 Software access to debug register.
- 110111 Exception Catch.
- 111011 Halting step, no syndrome.

All other values of STATUS are reserved.

**Accessing the EDSCR:**
EDSCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x088</td>
</tr>
</tbody>
</table>
G2.1.9 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

**Purpose**
Contains sampled values captured on reading EDPCSR.

**Usage constraints**
This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Configurations**
EDVIDSR is in the Core power domain.
Fields in this register reset to architecturally UNKNOWN values.
Required only if the OPTIONAL PC Sample-based Profiling Extension is implemented.

**Attributes**
EDVIDSR is a 32-bit register.

**Field descriptions**
The EDVIDSR bit assignments are:

```
<table>
<thead>
<tr>
<th>31 30 29 28 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
</tr>
<tr>
<td>E2</td>
</tr>
<tr>
<td>E3</td>
</tr>
</tbody>
</table>
```

**NS, bit [31]**
Reserved, RAO.

**E2, bit [30]**
Exception level 2 status sample. Indicates whether the most recent EDPCSR sample was associated with EL2.
If EL2 is not implemented, this bit is RES0.
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**E3, bit [29]**
Exception level 3 status sample. RES0.

**Bits [28:8]**
Reserved, RES0.

**VMID, bits [7:0]**
VMID sample. The value of VSCTRL.VMID associated with the most recent EDPCSR sample. If EDVIDSR.E2 == 1, this field is RAZ.
If EL2 is not implemented, this field is RES0.
When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Accessing the EDVIDSR:**

EDVIDSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x0A8</td>
</tr>
</tbody>
</table>
G2.1.10 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

**Usage constraints**

This register is accessible as follows:

<table>
<thead>
<tr>
<th>SLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Configurations**

PMAUTHSTATUS is in the Debug power domain.

This register is OPTIONAL, and is required for CoreSight compliance. ARM recommends that this register is implemented.

**Attributes**

PMAUTHSTATUS is a 32-bit register.

**Field descriptions**

The PMAUTHSTATUS bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>HNID HID SNID SID NSID NSNID</td>
</tr>
</tbody>
</table>

**Bits [31:12]**

Reserved, RES0.

**HNID, bits [11:10]**

Holds the same value as DBGAUTHSTATUS_EL1.HNID.

**HID, bits [9:8]**

Hyp invasive debug. Possible values of this field are:

- 00 Not implemented.
- Other values are reserved.

**SNID, bits [7:6]**

Secure non-invasive debug. Possible values of this field are:

- 00 Not implemented.
- Other values are reserved.

**SID, bits [5:4]**

Secure invasive debug. Possible values of this field are:

- 00 Not implemented.
Other values are reserved.

NSNID, bits [3:2]
Holds the same value as DBGAUTHSTATUS_EL1.NSNID.

NSID, bits [1:0]
Non-secure invasive debug. Possible values of this field are:
00 Not implemented.
Other values are reserved.

Accessing the PMAUTHSTATUS:
PMAUTHSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xF8</td>
</tr>
</tbody>
</table>
G2.1.11 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Usage constraints**

This register is accessible as follows:

<table>
<thead>
<tr>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EPMAD</th>
<th>SLK</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RW</td>
</tr>
</tbody>
</table>

**Configurations**

External register PMCR_EL0[6:0] is architecturally mapped to System register PMCR[6:0]. PMCR_EL0 is in the Core power domain. Some or all RW fields of this register have defined reset values. These apply on a Warm or Cold reset. The register is not affected by an External Debug reset. This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

**Attributes**

PMCR_EL0 is a 32-bit register.

**Field descriptions**

The PMCR_EL0 bit assignments are:

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 6 | 7 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RAZ/WI | RES0 | LC | D | C | P | E |

**Bits [31:11]**

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.

**LC, bit [6]**

Long cycle counter enable. Determines which PMCCNTR_EL0 bit generates an overflow recorded by PMOVSR[31].

0 Cycle counter overflow on increment that changes PMCCNTR_EL0[31] from 1 to 0.
1 Cycle counter overflow on increment that changes PMCCNTR_EL0[63] from 1 to 0.

ARM deprecates use of PMCR_EL0.LC = 0.

When this register has an architecturally defined reset value, this field resets to a value that is architecturally UNKNOWN.

**DP, bit [5]**

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

0 PMCCNTR_EL0, if enabled, counts when event counting is prohibited.
PMCCNTR_EL0 does not count when event counting is prohibited. If EL2 is not implemented, this field is RES0, otherwise it is an RW field.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

X, bit [4]
Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

0  Do not export events.
1  Export events where not prohibited.

This field enables the exporting of events over an event bus to another device, for example to an OPTIONAL trace macrocell. If the implementation does not include such an event bus then this field is RAZ/W1, otherwise it is an RW field.

In an implementation that includes an event bus, no events are exported when counting is prohibited. This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

When this register has an architecturally defined reset value, if this field is implemented as an RW field, it resets to 0.

D, bit [3]
Clock divider. The possible values of this bit are:

0  When enabled, PMCCNTR_EL0 counts every clock cycle.
1  When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR_EL0.D = 1.

When this register has an architecturally defined reset value, this field resets to 0.

C, bit [2]
Cycle counter reset. This bit is WO. The effects of writing to this bit are:

0  No action.
1  Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Resetting PMCCNTR_EL0 does not clear the PMCCNTR_EL0 overflow bit to 0.

P, bit [1]
Event counter reset. This bit is WO. The effects of writing to this bit are:

0  No action.
1  Reset all event counters, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

Resetting the event counters does not clear any overflow bits to 0.

If EL2 is implemented and HDCR.EPMAD is set to 1, a write of 1 to this bit does not reset event counters that HDCR.HPMN reserves for EL2 use.

E, bit [0]
Enable. The possible values of this bit are:

0  All counters that are accessible at EL1, including PMCCNTR, are disabled.
1  All counters that are accessible at EL1 are enabled by PMCNTENSET unless the value of HDCR.HPMD is 1, in which case event counting by the following counters is prohibited:
   •  Counters in the range of 0-n, where n is one less than the value of HDCR.
• HPMN. PMCCNTR if the value of PMCR.DP is 1.

This bit is RW.

If EL2 is implemented, this bit does not affect the operation of event counters that HDCR.HPMN reserves for EL2 use.

When this register has an architecturally defined reset value, this field resets to 0.

**Accessing the PMCR_EL0:**

PMCR_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE04</td>
</tr>
</tbody>
</table>
G2 Description of the Redefined External Debug Registers
G2.1 Redefined external debug registers
Part H

Architectural Pseudocode for ARMv8-R AArch32
Chapter H1
ARMv8-R AArch32 Pseudocode

This chapter contains pseudocode that describes many features of the ARMv8-R architecture. It contains the following sections:

- *Pseudocode limitations in Issue A.b* on page H1-228.
- *Pseudocode for AArch32 operation* on page H1-229.
- *Shared pseudocode* on page H1-287.
H1.1 Pseudocode limitations in Issue A.b

- Functions that address both AArch32 and AArch64 functionality might contain cases or comments that apply to only AArch64 state, EL3, Monitor mode, or Secure state and are therefore not applicable to the ARMv8-R AArch32 profile.

- Some functions and comments might contain information related to the Short-descriptor format that is not applicable to the ARMv8-R AArch32 profile.

- Tests might contain clauses that always return TRUE or FALSE in AArch32 state and therefore do not test the ARMv8-R AArch32 profile. For example in ARMv8-R AArch32:
  - UsingAArch32() always returns TRUE.
  - IsSecure always returns FALSE.

- Assertions that are not applicable to ARMv8-R AArch32 might be present. For example:
  - assert ELUsingAArch32(S1ValidationRegime()).

- Enumerations might contain values that are not applicable to ARMv8-R AArch32. For example:
  - AccType_PTW.
  - Fault_TLBConflict.
H1.2 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions that are listed in this section are identified as AArch32.FunctionName. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example aarch32/debug/breakpoint.

The top-level sections of the AArch32 pseudocode hierarchy are:

- aarch32/debug
- aarch32/exceptions
- aarch32/functions
- aarch32/translation

H1.2.1 aarch32/debug

aarch32/debug/VCRMatch/AArch32.VCRMatch

// AArch32.VCRMatch()
// ==============

boolean AArch32.VCRMatch(bits(32) vaddress)

if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
    // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
    match_word = Zeros(32);

    if vaddress<31:5> == ExcVectorBase()<31:5> then
        match_word<UInt(vaddress<4:2>)> = '1';
    
    mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0

    match_word = match_word AND DBGVCR AND mask;

    match = !IsZero(match_word);

    // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
    if !IsZero(match_word<4:3>) && DebugTarget() == PSTATE.EL then
        match = ConstrainUnpredictableBool();
    else
        match = FALSE;

    return match;

aarch32/debug/breakpoint/AArch32.BreakpointMatch

// AArch32.BreakpointMatch()
// ===============

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)

assert !UsingAArch32();
assert n <= UInt(DBGDIDR.BRPs);

enabled = DBGBCR[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC, linked, DBGBCR[n].LBN, isbreakpnt, ispriv);

(value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

if size == 4 then // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match the address of the
// second halfword of an instruction, but not the address of the first halfword, it is
// CONstrained UNPredictable whether or not this breakpoint generates a Breakpoint debug
// event.
(match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
if !value_match && match_i then
  value_match = ConstrainUnpredictableBool();
if value_mismatch && !mismatch_i then
  value_mismatch = ConstrainUnpredictableBool();

if vaddress<2> == '1' && DBGBCR[n].BAS == '1111' then
  if vaddress<2> == '1' && DBGBCR[n].BAS == '1111', then it is CONstrained
  // UNPredictable whether or not a Breakpoint debug event is generated for an instruction
  // at the address DBGVR[n]+2.
  if value_match then value_match = ConstrainUnpredictableBool();
  if !value_mismatch then value_mismatch = ConstrainUnpredictableBool();

match = value_match && state_match && enabled;
mismatch = value_mismatch && state_match && enabled;
return (match, mismatch);

aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.
(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

  // "n" is the identity of the breakpoint unit to match against
  // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
  // matching breakpoints.
  // "linked_to" is TRUE if this is a call from StateMatch for linking.

  // If a non-existent breakpoint then it is CONstrained UNPredictable whether this gives
  // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
  if n > UInt(DBGDIDR.BRPs) then
    (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs));
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
  // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
  // call from StateMatch for linking.)
  if DBGBCR[n].E == '0' then return (FALSE,FALSE);

  context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
  // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
  type = DBGBCR[n].BT;
  if (type IN {'011x','11xx'}) ||
    (type == '010x' && HaltOnBreakpointOrWatchpoint()) ||
    (type != '0x0x' && !HaveEL(EL2)) then
    (c, type) = ConstrainUnpredictableBits();
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
  // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
  // Determine what to compare against.
  match_addr = (type == '0x0x');
  mismatch = (type == '010x');
  match_vmid = (type == '10xx');
  match_cid = (type == 'x01x');
  linked = (type == 'xxx');
// 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 && (!linked || match_addr) then return (FALSE,FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked && linked && !match_addr then return (FALSE,FALSE);

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    assert byte IN [0,2];                     // "vaddress" is halfword aligned.
    byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
elsif match_cid then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
if match_vmid then
    vmid = VSCTRL.VMID;
    BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN [EL0,EL1] &&
        vmid == DBGBXVR[n]<7:0>);
    bvr_match_valid = (match_addr || match_cid);
    bxvr_match_valid = match_vmid;
match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return (match && !mismatch, !match && mismatch);

aarch32/debug/breakpoint/AArch32.StateMatch

// AArch32.StateMatch()
// ===========
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
    boolean isbreakpnt, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
// If parameters are set to a reserved type, behaves as either disabled or a defined type
if ((HMC:SSC:PxC) IN {'011xx','100x0','x01xx','x10xx','11101','1111x'} ||       // Reserved
    (HMC == '0' && PxC == '00' && !isbreakpnt) ||                             // Usr/Svc/Sys
    (HMC == '1' && !HaveEL(EL2))) then                            // No EL2
    (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
    assert c IN [ConstraintDISABLED, ConstraintUNKNOWN];
    if c == ConstraintDISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
    PL2_match = HaveEL(EL2) && HMC == '1';
    PL1_match = PxC<0> == '1';
    PL0_match = PxC<1> == '1';
    SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';
if SSU_match then
    priv_match = PSTATE.M IN [M32_User,M32_Svc,M32_System];
else:
    case PSTATE.EL of
        when EL3, EL1  priv_match = if ispriv then PL1_match else PL0_match;
        when EL2       priv_match = PL2_match;
        when EL0       priv_match = PL0_match;
    case SSC of
        when '00' security_state_match = TRUE;       // Both
when '01' security_state_match = !IsSecure(); // Non-secure only
when '10' security_state_match = IsSecure();   // Secure only
when '11' security_state_match = TRUE;        // Both

if linked then
  // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
  // it is CONstrained UNPREDICTABLE whether this gives no match, or LBN is mapped to some
  // UNKNOWN breakpoint that is context-aware.
  lbn = Uint(LBN);
  first_ctx_cmp = (Uint(DBGDIDR.BRPs) - Uint(DBGDIDR.CTX_CMPs));
  last_ctx_cmp = Uint(DBGDIDR.BRPs);
  if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
    (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
    assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    case c of
      when Constraint_DISABLED      return FALSE;      // Disabled
      when Constraint_NONE          linked = FALSE;    // No linking
  // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
  if linked then
    vaddress = bits(32) UNKNOWN;
    linked_to = TRUE;
    (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);

aarch32/debug/enables/AArch32.GenerateDebugExceptions

  // AArch32.GenerateDebugExceptions()
  // --------------------------------

  boolean AArch32.GenerateDebugExceptions()
  return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());

aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

  // AArch32.GenerateDebugExceptionsFrom()
  // -------------------------------------

  boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)
  if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
    return FALSE;

    enabled = from != EL2;
  return enabled;

aarch32/debug/pmu/AArch32.CheckForPMUOverflow

  // AArch32.CheckForPMUOverflow()
  // -----------------------------

  // Signal Performance Monitors overflow IRQ and CTI overflow events

  boolean AArch32.CheckForPMUOverflow()
  pmuirq = (PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSET<31> == '1');
  for n = 0 to Uint(PMCR.N) - 1
    if HaveEL(EL2) then
      hpmn = HDCR.HPMN;
      hpm = HDCR.HPME;
      E = (if n < Uint(hpmn) then PMCR.E else hpm); 
    else
      E = PMCR.E;
    if E == '1' && PMINTENSET<n> == '1' && PMOVSET<n> == '1' then pmuirq = TRUE;
SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

// The request remains set until the condition is cleared. (For example, an interrupt handler
// or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
return pmuirq;

aarch32/debug/pmu/AArch32.CountEvents

// AArch32.CountEvents()
// ===============
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
assert (n == 31 || n < UInt(PMCR.N));

// Event counting is disabled in Debug state
dbg = Halted();

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    hpmn = HDCR.HPMN;
    hpme = HDCR.HPME;
    E = (if n < UInt(hpmn) || n == 31 then PMCR.E else hpme);
else
    E = PMCR.E;

enabled = (E == '1' && PMCNTENSET<n> == '1');

if !IsSecure() then
    // Event counting is allowed unless all of:
    // * EL2 is implemented, and executing at EL2
    // * HPMD is implemented, PMNx is not reserved for Hyp, and HDCR.HPMD == 1
    if PSTATE.EL == EL2 && (n < UInt(hpmn) || n == 31) then
        prohibited = (HDCR.HPMD == '1');
    else
        prohibited = FALSE;
else
    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    prohibited = FALSE;

// For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = (if n == 31 then PMCCFILTR<31:27> else PMEVTYPER[n]<31:27>);

H = if !HaveEL(EL2) then '0' else filter<0>;
P = filter<4>;  U = filter<3>
if !IsSecure() && HaveEL(EL3) then
    P = P EOR filter<2>;  U = U EOR filter<1>

case PSTATE.EL of
    when EL0      filtered = U == '1';
    when EL1,EL3  filtered = P == '1';
    when EL2      filtered = H == '0';
return !debug && enabled && !prohibited && !filtered;

aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

// AArch32.EnterHypModeInDebugState()
// ================================
// Take an exception in Debug state to Hyp mode.
H1 ARMv8-R AArch32 Pseudocode
H1.2 Pseudocode for AArch32 operation

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

AArch32.ReportHypEntry(exception);
AArch32.WriteMode(M32_Hyp);
SPSR[] = bits(32) UNKNOWN;
ELR_hyp = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.(S,S,A,I,F) are not observable so behave as UNKNOWN.
PSTATE.T = '1';                             // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = HSCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
EDSCR.ERR = '1';
UpdateEDSCRFields();
EndOfInstruction();

aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

// AArch32.EnterModeInDebugState()
// ==============================================================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.
AArch32.EnterModeInDebugState(bits(5) target_mode)
assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

AArch32.WriteMode(target_mode);
SPSR[] = bits(32) UNKNOWN;
R[14] = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.(S,S,A,I,F) are not observable so behave as UNKNOWN.
PSTATE.T = '1';                             // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = SCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
EDSCR.ERR = '1';
UpdateEDSCRFields();                        // Update EDSCR processor state flags.
EndOfInstruction();

aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

// AArch32.WatchpointByteMatch()
// ==============================================================

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
// DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
  byte_select_match = ConstrainUnpredictableBool();
else
  LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
  if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
    byte_select_match = ConstrainUnpredictableBool();
bottom = 3;                                        // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
    (c, mask) = ConstrainUnpredictableInteger(3, 31);
    assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    case c of
        when Constraint_DISABLED return FALSE;            // Disabled
        when Constraint_NONE      mask = 0;                // No masking
    // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

if mask > bottom then
    WVR_match = (vaddress<31:mask> == DBGWVR<n><31:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR<n><mask-1:bottom>) then
        WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<31:bottom> == DBGWVR<n><31:bottom>;
    return WVR_match && byte_select_match;

aarch32/debug/watchpoint/AArch32.WatchpointMatch

// AArch32.WatchpointMatch()
// ================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
    boolean iswrite)
    assert ELUsingAArch32(S1ValidationRegime());
    assert n <= UInt(DBGDIDR.WRPs);
    // "ispriv" is FALSE for LDRT/STRRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR<n>.E == '1';
    linked = DBGWCR<n>.WT == '1';
    isbreakpnt = FALSE;
    state_match = AArch32.StateMatch(DBGWCR<n>.SSC, DBGWCR<n>.HMC, DBGWCR<n>.PAC,
        linked, DBGWCR<n>.LBN, isbreakpnt, ispriv);
    ls_match = (DBGWCR<n>.LSC<(if iswrite then 1 else 0)> == '1');
    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);
    return value_match && state_match && ls_match && enabled;

H1.2.2  aarch32/exceptions

aarch32/exceptions/aborts/AArch32.Abort

// AArch32.Abort()
// ==============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    if fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
aarch32/exceptions/aborts/AArch32.AbortSyndrome

// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception type, FaultRecord fault, bits(32) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type == Exception_DataAbort;

    exception.syndrome = FaultSyndrome(d_side, fault);

    exception.vaddress = ZeroExtend(vaddress);

    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.ipaddress = ZeroExtend(fault.ipaddress);
    else
        exception.ipavalid = FALSE;

    return exception;

aarch32/exceptions/aborts/AArch32.CheckPCAlignment

// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

bits(32) pc = ThisInstrAddr();

if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then

    // Generate an Alignment fault Prefetch Abort exception
    vaddress = pc;
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    secondstage = FALSE;
    AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));

aarch32/exceptions/aborts/AArch32.ReportDataAbort

// AArch32.ReportDataAbort()
// =========================

// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor format. Normally, the current translation table format determines the format. For an abort from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if any of the following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * The abort is synchronous and either:
    //   - It is taken from Hyp mode.
    //   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    // In the current implementation, PE supports PMSA at both EL1 and EL2.
    // v8R PMSA MPU follow Long-descriptor format
    long_format = TRUE;
    d_side = TRUE;

    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
if (!long_format &&
    boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
  i_syndrome = syndrome;
  syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
else
  i_syndrome = bits(32) UNKNOWN;
if route_to_monitor then
  IFSR_S = i_syndrome;
else
  IFSR = i_syndrome;
if route_to_monitor then
  DFSR_S = syndrome;
  DFAR_S = vaddress;
else
  DFSR = syndrome;
  DFAR = vaddress;
return;

aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.
AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
  // In the current implementation, PE supports PMSA at both EL1 and EL2.
  // v8R PMSA MPU follow long-descriptor format
  long_format = TRUE;
  d_side = FALSE;
  if long_format then
    fsr = AArch32.FaultStatusLD(d_side, fault);
  else
    fsr = AArch32.FaultStatusSD(d_side, fault);
if route_to_monitor then
  IFSR_S = fsr;
  IFAR_S = vaddress;
else
  IFSR = fsr;
  IFAR = vaddress;
return;

aarch32/exceptions/aborts/AArch32.TakeDataAbortException

// AArch32.TakeDataAbortException()
// ================================
AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
  route_to_monitor = FALSE;
  route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                  (HCR.TGE == '1' || IsSecondStage(fault) ||
                   (IsDebugException(fault) && HDCR.TDE == '1')));
  bits(32) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x10;
  lr_offset = 8;
if IsDebugException(fault) then
  DBGDSCRext.MOE = fault.debugmoe;
if PSTATE.EL == EL2 || route_to_hyp then
  exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

// AArch32.TakePrefetchAbortException()
// ====================================
AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = FALSE;
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE_EL IN {EL0,EL1} &&
        (HCR.TGE == '1' || IsSecondStage(fault) ||
            (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;
    if IsDebugException(fault) then
        if PSTATE_EL == EL2 || route_to_hyp then
            if fault.type == Fault_Alignment then             // PC Alignment fault
                exception = ExceptionSyndrome(Exception_PCAlignment);
                exception.vaddress = ThisInstrAddr();
            else
                exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
            end if
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        end if
    else
        if PSTATE_EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        end if
    end if

aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException

// AArch32.TakePhysicalFIQException()
// ===================================
AArch32.TakePhysicalFIQException()

// Take an enabled physical IRQ exception.

route_to_monitor = FALSE;
route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE_EL IN {EL0,EL1} &&
        (HCR.TGE == '1' || HCR.IMO == '1'));

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;
if PSTATE_EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_IRQ);
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException

// AArch32.TakePhysicalIRQException()
// ================================
AArch32.TakePhysicalIRQException()
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 4;
if PSTATE.EL == EL2 || route_to_hyp then
  exception = ExceptionSyndrome(Exception_IRQ);
  AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
  AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException

// AArch32.TakePhysicalSErrorException()
// -----------------------------------

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, boolean syndrome_valid,
  bits(24) full_syndrome)

route_to_monitor = FALSE;
route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1}) &&
  (HCR.TGE == '1' || HCR.AMO == '1');
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;

fault = AArch32.AsynchExternalAbort(parity, extflag);
vaddress = bits(32) UNKNOWN;
if PSTATE.EL == EL2 || route_to_hyp then
  exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
  AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
  AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

// AArch32.TakeVirtualFIQException()
// -----------------------------------

AArch32.TakeVirtualFIQException()

assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
assert HCR.TGE == '0' && HCR.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;
AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

// AArch32.TakeVirtualIRQException()
// -----------------------------------

AArch32.TakeVirtualIRQException()

assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
assert HCR.TGE == '0' && HCR.IMO == '1';
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 4;
AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```c
// AArch32.TakeVirtualSErrorException()
// ---------------------------------------
AArch32.TakeVirtualSErrorException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    route_to_monitor = FALSE;
    bits(32) preferred_exception_return = ThisInstrAddr();
    lr_offset = 0x10;
    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    extflag = bit IMPLEMENTATION_DEFINED "Virtual SError ExT bit";
    fault = AArch32.AsynchExternalAbort(parity, extflag);
    HCR.VA = '0';
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

```c
// AArch32.SoftwareBreakpoint()
// -----------------------------
AArch32.SoftwareBreakpoint(bits(16) immediate)
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH; // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;
    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

```c
constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
```

```c
// AArch32.ExceptionClass()
// -------------------------
// Return the Exception Class and Instruction Length fields for reported in HSR
(integer,bit) AArch32.ExceptionClass(Exception type)
    il = if ThisInstrLength() == 32 then '1' else '0';
    case type of
        when Exception_Uncategorized ec = 0x00; il = '1';
        when Exception_WFxTrap ec = 0x01;
        when Exception_CPI15RTTrap ec = 0x03;
        when Exception_CPI15RRTTrap ec = 0x04;
        when Exception_CPI14RTTrap ec = 0x05;
        when Exception_CPI14DTTrap ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap ec = 0x08;
        when Exception_CPI14RRTTrap ec = 0x0C;
```
when Exception_IllegalState         ec = 0x0E; il = '1';
when Exception_SupervisorCall       ec = 0x11;
when Exception_HypervisorCall       ec = 0x12;
when Exception_MonitorCall          ec = 0x13;
when Exception/InstructionAbort      ec = 0x20; il = '1';
when Exception_PCAlignment          ec = 0x22; il = '1';
when Exception_DataAbort            ec = 0x24;
when Exception_FPTrappedException   ec = 0x28;
otherwise                           Unreachable();
if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
  ec = ec + 1;
return (ec,il);

aarch32/exceptions/exceptions/AArch32.ReportHypEntry

// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.
AArch32.ReportHypEntry(ExceptionRecord exception)

Exception type = exception.type;
(ec,il) = AArch32.ExceptionClass(type);
iss = exception.syndrome;

// IL is not valid for Data Abort exceptions without valid instruction syndrome information
if ec IN {0x24,0x25} && iss<24> == '0' then
  il = '1';
HSR = ec<5:0>:il:iss;
if type IN {Exception/InstructionAbort, Exception_PCAlignment} then
  HIFAR = exception.vaddress<31:0>;
  HDFAR = bits(32) UNKNOWN;
elsif type == Exception_DataAbort then
  HIFAR = bits(32) UNKNOWN;
  HDFAR = exception.vaddress<31:0>;
if exception.ipavalid then
  HPFAR<31:4> = exception.ipaddress<39:12>;
else
  HPFAR<31:4> = bits(28) UNKNOWN;
return;

aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);

aarch32/exceptions/exceptions/AArch32.TakeReset

// AArch32.TakeReset()
// ===============
// Reset into AArch32 state
AArch32.TakeReset(boolean cold_reset)
  assert HighestELUsingAArch32();
  // Enter the highest implemented Exception level in AArch32 state
  if HaveEL(EL2) then
    AArch32.WriteMode(M32_Hyp);
else

    AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.A<4,3,0> = '111';       // All asynchronous exceptions masked
    PSTATE.IT = '00000000';       // IT block state reset
    PSTATE.T = SCTLR.TE;          // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE;          // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector

    rv = RVBAR<31:1>:1:'0';
    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType_UNKNOWN);

aarch32/exceptions/exceptions/ExcVectorBase

    // ExcVectorBase()
    // ===============

    bits(32) ExcVectorBase()

    return VBAR;

aarch32/exceptions/ieeefp/AArch32.FPTrappedException

    // AArch32.FPTrappedException()
    // ===============

    AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    AArch32.TakeUndefInstrException();

aarch32/exceptions/syscalls/AArch32.CallHypervisor

    // AArch32.CallHypervisor()
    // ===============

    // Performs a HVC call

    AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    AArch32.TakeHVCException(immediate);
aarch32/exceptions/syscalls/AArch32.CallSupervisor

// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
        AArch32.TakeSVCException(immediate);

aarch32/exceptions/syscalls/AArch32.TakeHVCException

// AArch32.TakeHVCException()
// =========================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

AArch32.ITAdvance();
bits(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;

exception = ExceptionSyndrome(Exception_HypervisorCall);
exception.syndrome<15:0> = immediate;

if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);

aarch32/exceptions/syscalls/AArch32.TakeSVCException

// AArch32.TakeSVCException()
// =========================

AArch32.TakeSVCException(bits(16) immediate)
    AArch32.ITAdvance();
    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

bits(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;

if PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/takeexception/AArch32.EnterHypMode

// AArch32.EnterHypMode()
// =======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
integer vect_offset)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
spsr = GetPSRFromPSTATE();
if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
  AArch32.ReportHypEntry(exception);
AArch32.WriteMode(M32_Hyp);
SPSR[] = spsr;
ELR_hyp = preferred_exception_return;
PSTATE.T = HSCTRL.TE;  // PSTATE.J is RES0
PSTATE.SS = '0';
PSTATE.A = '1';
PSTATE.I = '1';
PSTATE.F = '1';
PSTATE.E = HSCTRL.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
BranchTo(HVBAR + vect_offset, BranchType_UNKNOWN);
EndOfInstruction();

aarch32/exceptions/takeexception/AArch32.EnterMode

  // AArch32.EnterMode()
  // ===================
  // Take an exception to a mode other than Monitor and Hyp mode.
AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
spsr = GetPSRFromPSTATE();
AArch32.WriteMode(target_mode);
SPSR[] = spsr;
R[14] = preferred_exception_return + lr_offset;
PSTATE.T = SCTLR.TE;  // PSTATE.J is RES0
PSTATE.SS = '0';
if target_mode == M32_FIQ then
  PSTATE.<A,I,F> = '111';
else if target_mode IN {M32_Abort, M32_IRQ} then
  PSTATE.<A,I> = '11';
else
  PSTATE.I = '1';
PSTATE.E = SCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
BranchTo(ExcVectorBase() + vect_offset, BranchType_UNKNOWN);
EndOfInstruction();

aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrap

  // AArch32.AArch32SystemAccessTrap()
  // =================================
  // Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.
AArch32.AArch32SystemAccessTrap(bits(2) target_el, bits(32) instr)
assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
assert target_el IN {EL1,EL2};
if target_el == EL2 then
  exception = AArch32.AArch32SystemAccessTrapSyndrome(instr);
  AArch32.TakeHypTrapException(exception);
else
  AArch32.TakeUndefInstrException();
aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrapSyndrome

// AArch32.AArch32SystemAccessTrapSyndrome()
// =========================================
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.AArch32SystemAccessTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' & instr<4> == '1' & instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
        when 10  exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 14  exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 15  exception = ExceptionSyndrome(Exception_CP15RTTrap);
        otherwise  Unreachable();

        iss<19:17> = instr<7:5>;      // opc2
        iss<16:14> = instr<23:21>;    // opc1
        iss<13:10> = instr<19:16>;    // CRn
        iss<8:5>  = instr<15:12>;    // Rt
        iss<4:1>  = instr<3:0>;      // CRm
    elsif instr<27:21> == '1100010' & instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
        when 14  exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        when 15  exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        otherwise  Unreachable();

        iss<19:16> = instr<7:4>;      // opc1
        iss<13:10> = instr<19:16>;    // Rt2
        iss<8:5>  = instr<15:12>;    // Rt
        iss<4:1>  = instr<3:0>;      // CRm
    elsif instr<27:25> == '110' & instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;

        exception = ExceptionSyndrome(Exception_CP14DTTrap);

        iss<19:12> = instr<7:0>;      // imm8
        iss<4>    = instr<23>;        // U
        iss<2:1>  = instr<24:21>;     // P,W
        if instr<19:16> == '1111' then // Literal addressing
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
        else
            iss<8:5> = instr<19:16>;    // Rn
            iss<3>   = '0';
        end...
    else
        Unreachable();

    iss<0> = instr<20>;             // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0> = iss;

    return exception;

aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)

    cpacr_asedis = CPACR.ASEDIS;
    cpacr_cpi0 = CPACR.cpi0;}
if HaveEL(EL3) && !IsSecure() then
    // Check if access disabled in NSACR
    if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
    if NSACR.cp10 == '0' then cpacr_cp10 = '00';

if PSTATE.EL != EL2 then
    // Check if Advanced SIMD disabled in CPACR
    if advsimd && cpacr_asedis == '1' then UNDEFINED;
    // Check if access disabled in CPACR
    case cpacr_cp10 of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then UNDEFINED;
    // If required, check FPEXC enabled bit.
    if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

AArch32.CheckFPAdvSIMDTrap(advsimd); // Also check against HCPTR and CPTR_EL3

aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

// AArch32.CheckFPAdvSIMDTrap()
// ================
// Check against CPTR_EL2 and CPTR_EL3.
AArch32.CheckFPAdvSIMDTrap(boolean advsimd)

    if HaveEL(EL2) && !IsSecure() then
        hcptr_tase = HCPTR.TASE;
        hcptr_cp10 = HCPTR.TCP10;
    if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
        // Check if access disabled in NSACR
        if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
        if NSACR.cp10 == '0' then hcptr_cp10 = '1';
    // Check if access disabled in HCPTR
    if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        if advsimd then
            exception.syndrome<5> = '1';
        else
            exception.syndrome<5> = '0';
        exception.syndrome<3:0> = '1010'; // coproc field, always 0xA
        if PSTATE.EL == EL2 then
            AArch32.TakeUndefInstrException(exception);
        else
            AArch32.TakeHypTrapException(exception);
    return;

aarch32/exceptions/traps/AArch32.CheckForWFxTrap

// AArch32.CheckForWFxTrap()
// ================
// Check for trap on WFE or WFI instruction
AArch32.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);
    case target_el of
when EL1 trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
when EL2 trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
if trap then
    if target_el == EL2 then
        exception = ExceptionSyndrome(Exception_WFxTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        exception.syndrome<0> = if is_wfe then '1' else '0';
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();

aarch32/exceptions/traps/AArch32.CheckITEnabled

// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.
AArch32.CheckITEnabled(bits(4) mask)
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = SCTLR.ITD;
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;
        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];
        if next_instr IN ['11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx', '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'] then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;
    return;

aarch32/exceptions/traps/AArch32.CheckIllegalState

// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.
AArch32.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';
        preferred_exception_return = ThisInstrAddr();
        vector_offset = 0x04;
        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vector_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

// AArch32.CheckSETENDEnabled()
// =====================================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
if PSTATE.EL == EL2 then
  setend_disabled = HSCTLR.SED;
else
  setend_disabled = SCTLR.SED;
if setend_disabled == '1' then
  UNDEFINED;
  return;

aarch32/exceptions/traps/AArch32.TakeHypTrapException

// AArch32.TakeHypTrapException()
// ===============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x14;
AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);

aarch32/exceptions/traps/AArch32.TakeUndefInstrException

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
exception = ExceptionSyndrome(Exception_Uncategorized);
AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)
route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x04;
lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
if PSTATE.EL == EL2 then
  AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
elsif route_to_hyp then
  AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
  AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/traps/AArch32.UndefinedFault

// AArch32.Undefaulted()
// =======================
AArch32.UndefinedFault()

AArch32.TakeUndefInstrException();

H1.2.3 aarch32/functions

aarch32/functions/aborts/AArch32.CreateFaultRecord

// AArch32.CreateFaultRecord()
// ===========================

FaultRecord AArch32.CreateFaultRecord(Fault type, bits(32) ipaddress, bits(4) domain,
integer level, AccType acctype, boolean write, bit extflag,
bits(4) debugmoe, boolean secondstage, boolean s2fs1walk)

FaultRecord fault;
fault.type = type;
if (type != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk &&
AArch32.DomainValid(type, level)) then
fault.domain = domain;
else
fault.domain = bits(4) UNKNOWN;
fault.debugmoe = debugmoe;
fault.ipaddress = ZeroExtend(ipaddress);
fault.level = level;
fault.acctype = acctype;
fault.write = write;
fault.extflag = extflag;
fault.secondstage = secondstage;
fault.s2fs1walk = s2fs1walk;
return fault;

aarch32/functions/aborts/AArch32.DomainValid

// AArch32.DomainValid()
// =============

// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault type, integer level)
assert type != Fault_None;
case type of
when Fault_Domain
return TRUE;
when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
return level == 2;
otherwise
return FALSE;

aarch32/functions/aborts/AArch32.FaultStatusLD

// AArch32.FaultStatusLD()
// ==============

// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
assert fault.type != Fault_None;
bits(32) fsr = Zeros();
if d_side then
if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
fsr<13> = '1'; fsr<11> = '1';
else
    fsr<11> = if fault.write then '1' else '0';
if IsExternalAbort(fault) then fsr<12> = fault.extflag;
fsr<9> = '1';
fsr<5:0> = EncodeLDFSC(fault.type, fault.level);
return fsr;

aarch32/functions/aborts/AArch32.FaultStatusSD

// AArch32.FaultStatusSD()
//=----------------------
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;
    bits(32) fsr = Zeros();
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then fsr<12> = fault.extflag;
        fsr<9> = '0';
        fsr<10,3:0> = EncodeSDFSC(fault.type, fault.level);
        if d_side then
            fsr<7:4> = fault.domain;   // Domain field (data fault only)
    return fsr;

aarch32/functions/aborts/EncodeSDFSC

// EncodeSDFSC()
//=------------
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault type, integer level)
    bits(5) result;
    case type of
        when Fault_AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
result = if level == 1 then '11100' else '11110';
when Fault.AsyncParity
result = '11000';
when Fault.AsyncExternal
result = '10110';
when Fault.Debug
result = '00010';
when Fault.TLBConflict
result = '10000';
when Fault.Lockdown
result = '10100'; // IMPLEMENTATION DEFINED
when Fault.Exclusive
result = '10101'; // IMPLEMENTATION DEFINED
when Fault.ICacheMaint
result = '00100';
otherwise
Unreachable();
return result;

aarch32/functions/common/A32ExpandImm

// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);
return imm32;

aarch32/functions/common/A32ExpandImm_C

// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

unrotated_value = ZeroExtend(imm12<7:0>, 32);
(imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);
return (imm32, carry_out);

aarch32/functions/common/DecodeImmShift

// DecodeImmShift()  // =============

(SRTYPE, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

case type of
when '00'
shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
when '01'
shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
when '10'
shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
when '11'
if imm5 == '00000'
shift_t = SRTYPE_RRX; shift_n = 1;
else
shift_t = SRTYPE_ROR; shift_n = UInt(imm5);
return (shift_t, shift_n);
aarch32/functions/common/DecodeRegShift

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00' shift_t = SRType_LSL;
        when '01' shift_t = SRType_LSR;
        when '10' shift_t = SRType_ASR;
        when '11' shift_t = SRType_ROR;
    return shift_t;

aarch32/functions/common/RRX

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;

aarch32/functions/common/RRX_C

// RRX_C()
// ========

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

aarch32/functions/common/SRType

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};

aarch32/functions/common/Shift

// Shift()
// ========

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;

aarch32/functions/common/Shift_C

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
    }
(result, carry_out) = ROR_C(value, amount);
when SRType_RRX
    (result, carry_out) = RRX_C(value, carry_in);

return (result, carry_out);

aarch32/functions/common/T32ExpandImm

// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;

aarch32/functions/common/T32ExpandImm_C

// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
                carry_out = carry_in;
        else
            unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
            (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

        return (imm32, carry_out);

aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

// AArch32.CheckCP15InstrCoarseTraps()
// ===================================

// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if HaveEL(EL2) & & !IsSecure() & & PSTATE.EL IN {EL0,EL1} then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) & & HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' & & nreg == 1 & &
            (CRn == 9 & & CRm IN {0,1,2, 5,6,7,8 }) ||
            (CRn == 10 & & CRm IN {0,1, 4, 8 }) ||
            (CRn == 11 & & CRm IN {0,1,2,3,4,5,6,7,8,15})) then
            return TRUE;

    return FALSE;
AArch32.CheckSystemAccess(integer cp_num, bits(32) instr)
assert cp_num == UInt(instr<11:8>) && (cp_num IN {14,15});

// Decode the AArch32 System register access instruction
if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then      // MRC/MCR
  cprt = TRUE;  cpdt = FALSE;  nreg = 1;
  opc1 = UInt(instr<23:21>);
  opc2 = UInt(instr<7:5>);
  CRn  = UInt(instr<19:16>);
  CRm  = UInt(instr<3:0>);
elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then                   // MRRC/MCRR
  cprt = TRUE;  cpdt = FALSE;  nreg = 2;
  opc1 = UInt(instr<7:4>);
  CRm  = UInt(instr<3:0>);
elsif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then   // LDC/STC
  cprt = FALSE;  cpdt = TRUE;  nreg = 0;
  opc1 = 0;
  CRn  = UInt(instr<15:12>);
else
  allocated = FALSE;

// Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
// returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
if cp_num == 14 then
// LDC and STC only supported for c5 in CP14 encoding space
if cpdt && CRn != 5 then
  allocated = FALSE;
else
// Coarse-grained decode of CP14 based on opc1 field
  case opc1 of
    when 0     allocated = CP14DebugInstrDecode(instr);
    when 1     allocated = CP14TraceInstrDecode(instr);
    when 7     allocated = CP14JazelleInstrDecode(instr);    // JIDR only
    otherwise  allocated = FALSE;          // All other values are unallocated
  endcase
elsif cp_num == 15 then
// LDC and STC not supported in CP15 encoding space
if !cpdt then
  allocated = FALSE;
else
  // Coarse-grain traps to EL2 have a higher priority than exceptions generated because
  // the access instruction is UNDEFINED
  if AArch32.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
    // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
    // Non-secure User mode is UNDEFINED when the trap is disabled, then it is
    // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
    // when the trap is enabled.
    if PSTATE.EL == EL0 && !IsSecure() && !allocated then
      if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
        UNDEFINED;
        AArch32.AArch32SystemAccessTrap(EL2, instr);
      else
        allocated = FALSE;
        if !allocated then
          UNDEFINED;
          if !allocated then
            UNDEFINED;
      endcase
    endif
  endif
else
  allocated = FALSE;
if !allocated then
  UNDEFINED;
// If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
AArch32.CheckSystemAccessTraps(instr);
return;

aarch32/functions/coproc/AArch32.CheckSystemAccessTraps

// Check for configurable disables or traps to a higher EL of an System register access.
AArch32.CheckSystemAccessTraps(bits(32) instr);

aarch32/functions/coproc/CP14DebugInstrDecode

// Decodes an accepted access to a debug System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14DebugInstrDecode(bits(32) instr);

aarch32/functions/coproc/CP14JazelleInstrDecode

// Decodes an accepted access to a Jazelle System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14JazelleInstrDecode(bits(32) instr);

aarch32/functions/coproc/CP14TraceInstrDecode

// Decodes an accepted access to a trace System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14TraceInstrDecode(bits(32) instr);

aarch32/functions/coproc/CP15InstrDecode

// Decodes an accepted access to a System register in the CP15 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP15InstrDecode(bits(32) instr);

aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

// AArch32.ExclusiveMonitorsPass()
// ==================================

// Return TRUE if the Exclusive Monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusive Monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

acctype = AccType_ATOMIC;
iswrite = TRUE;
aligned = (address == Align(address, size));

if !aligned then
  secondstage = FALSE;
  AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
if !passed then
  return FALSE;
memaddrdesc = AArch32.ValidateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
if passed then
    ClearExclusiveLocal(ProcessorID());
    if memaddrdesc.memattrs.shareable then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

return passed;

aarch32/functions/exclusive/AArch32.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
bool AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);

aarch32/functions/exclusive/AArch32.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);

aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

// AArch32.SetExclusiveMonitors()
// =============================
// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.
AArch32.SetExclusiveMonitors(bits(32) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));
    memaddrdesc = AArch32.ValidateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    AArch32.MarkExclusiveVA(address, ProcessorID(), size);

aarch32/functions/float/CheckAdvSIMDEnabled

// CheckAdvSIMDEnabled()
// =====================
CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;
AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
// Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

// Make temporary copy of D registers
// _Dclone[] is used as input data for instruction pseudocode
for i = 0 to 31
    _Dclone[i] = D[i];

return;

aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

// CheckAdvSIMDOrVFPEnabled()
// ==========================
CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
// Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
return;

aarch32/functions/float/CheckCryptoEnabled32

// CheckCryptoEnabled32()
// ======================
CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
// Return from CheckAdvSIMDEnabled() occurs only if access is permitted
return;

aarch32/functions/float/CheckVFPEnabled

// CheckVFPEnabled()
// =================
CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
// Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
return;

aarch32/functions/float/FPHalvedSub

// FPHalvedSub()
// =============
bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN (32,64);
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
    if inf1 && inf2 && sign1 == sign2 then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
        result = FPInfinity('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
        result = FPInfinity('1');
    elsif zero1 && zero2 && sign1 != sign2 then
        result = FPZero(sign1);

else
    result_value = (value1 - value2) / 2.0;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr);
    return result;

aarch32/functions/float/FPRSqrtStep

// FPRSqrtStep()
// =============

bits(32) FPRSqrtStep(bits(32) op1, bits(32) op2)
  FPCRType fpcr = StandardFPSCRValue();
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
    product;
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero('0');
    else
      product = FPMul(op1, op2, fpcr);
  result = FPHalvedSub(FPThree('0'), product, fpcr);
  return result;

aarch32/functions/float/FPRecipStep

// FPRecipStep()
// =============

bits(32) FPRecipStep(bits(32) op1, bits(32) op2)
  FPCRType fpcr = StandardFPSCRValue();
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
    product;
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero('0');
    else
      product = FPMul(op1, op2, fpcr);
  result = FPSub(FPTwo('0'), product, fpcr);
  return result;

aarch32/functions/float/StandardFPSCRValue

// StandardFPSCRValue()
// ==========

FPCRType StandardFPSCRValue()
  return '00000' : FPSCR.AHP : '11000000000000000000000000';

aarch32/functions/memory/AArch32.CheckAlignment

// AArch32.CheckAlignment()
// ==========

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
boolean iswrite)
    if PSTATE.EL == EL2 then
        A = HSCTLR.A;
    else
        A = SCTLR.A;
    aligned = (address == Align(address, alignment));

    // AccType_VEC is used for SIMD element alignment checks only
    check = (acctype == AccType_ATOMIC || acctype == AccType_ORDERED || acctype == AccType_VEC || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));
    return aligned;

aarch32/functions/memory/AArch32.MemSingle

// AArch32.MemSingle[] - non-assignment (read) form
// 剥削短:::// 本 atomic, little-endian read of 'size' bytes.
bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);
    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch32.ValidateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);
    // Memory array access
    value = _Mem[memaddrdesc, size, acctype];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// 本 atomic, little-endian write of 'size' bytes.
AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);
    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch32.ValidateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);
    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
// Memory array access
_mem[memaddrdesc, size, acctype] = value;
return;

aarch32/functions/memory/Hint_PreloadData
Hint_PreloadData(bits(32) address);

aarch32/functions/memory/Hint_PreloadDataForWrite
Hint_PreloadDataForWrite(bits(32) address);

aarch32/functions/memory/Hint_PreloadInstr
Hint_PreloadInstr(bits(32) address);

aarch32/functions/memory/MemA
// MemA[] - non-assignment form
//=-----------------------------------------
bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
//=-----------------------------------------
MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;

aarch32/functions/memory/MemO
// MemO[] - non-assignment form
//=-----------------------------------------
bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType_ORDERED;
    return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
//=-----------------------------------------
MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ORDERED;
    Mem_with_type[address, size, acctype] = value;
    return;

aarch32/functions/memory/MemU
// MemU[] - non-assignment form
//=-----------------------------------------
bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
//=-----------------------------------------
MemU[bits(32) address, integer size] = bits(8*size) value


```c
acctype = AccType_NORMAL;
Mem_with_type[address, size, acctype] = value;
return;
```

```c
aarch32/functions/memory/MemU_unpriv

// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
```

```c
aarch32/functions/memory/Mem_with_type

// Mem_with_type[] - non-assignment (read) form
// ============================================

// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;
    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];
    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    c = ConstrainUnpredictable();
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;
    for i = 1 to size-1
        value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];
    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =========================================

// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;
    if BigEndian() then
        value = BigEndianReverse(value);
```
aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

if !aligned then
    assert size > 1;
    AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    c = ConstrainUnpredictable();
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;

    for i = 1 to size-1
        AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;

return;

aarch32/functionsregisters/AArch32.ResetGeneralRegisters

// AArch32.ResetGeneralRegisters()
// ------------------------------
AArch32.ResetGeneralRegisters()

for i = 0 to 7
    R[i] = bits(32) UNKNOWN;
for i = 8 to 12
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;  // No R14_hyp
for i = 13 to 14
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
    Rmode[i, M32_Svc] = bits(32) UNKNOWN;
    Rmode[i, M32_Abort] = bits(32) UNKNOWN;
    Rmode[i, M32_Undef] = bits(32) UNKNOWN;
if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

return;

aarch32/functionsregisters/AArch32.ResetSIMDFPRegisters

// AArch32.ResetSIMDFPRegisters()
// -------------------------------
AArch32.ResetSIMDFPRegisters()

for i = 0 to 15
    Q[i] = bits(128) UNKNOWN;

return;

aarch32/functionsregisters/AArch32.ResetSpecialRegisters

// AArch32.ResetSpecialRegisters()
// --------------------------------
AArch32.ResetSpecialRegisters()

// AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
SPSR_abt = bits(32) UNKNOWN;
SPSR_und = bits(32) UNKNOWN;
if HaveEL(EL2) then
  SPSR_hyp = bits(32) UNKNOWN;
  ELR_hyp = bits(32) UNKNOWN;
if HaveEL(EL3) then
  SPSR_mon = bits(32) UNKNOWN;

// External debug special registers
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
return;

aarch32/functions/registers/AArch32.ResetSystemRegisters
AArch32.ResetSystemRegisters(boolean cold_reset);

aarch32/functions/registers/ALUExceptionReturn

// ALUExceptionReturn()
// ================

ALUExceptionReturn(bits(32) address)
  if PSTATE.EL == EL2 then
    UNDEFINED;
  elsif PSTATE.M IN {M32_User,M32_System} then
    UNPREDICTABLE;          // UNDEFINED or NOP
  else
    AArch32.ExceptionReturn(address, SPSR[]);

aarch32/functions/registers/ALUWritePC

// ALUWritePC()
// =============

ALUWritePC(bits(32) address)
  if CurrentInstrSet() == InstrSet_A32 then
    BXWritePC(address);
  else
    BranchWritePC(address);

aarch32/functions/registers/BXWritePC

// BXWritePC()
// ===========

BXWritePC(bits(32) address)
  if address<0> == '1' then
    SelectInstrSet(InstrSet_T32);
    address<0> = '0';
  else
    SelectInstrSet(InstrSet_A32);
    // For branches to an unaligned PC counter in A32 state, the processor takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address<1> == '1' & ConstrainUnpredictableBool() then
      address<1> = '0';
      BranchTo(address, BranchType_UNKNOWN);
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
  if CurrentInstrSet() == InstrSet_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
  BranchTo(address, BranchType_UNKNOWN);

// D[] - non-assignment form
// =========================

bits(64) D[integer n]
  assert n >= 0 && n <= 31;
  base = (n MOD 2) * 64;
  return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
  assert n >= 0 && n <= 31;
  base = (n MOD 2) * 64;
  _V[n DIV 2]<base+63:base> = value;
  return;

// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
  assert n >= 0 && n <= 31;
  return _Dclone[n];

// LR - assignment form
// ====================

LR = bits(32) value
  R[14] = value;
  return;

// LR - non-assignment form
// ========================

bits(32) LR
  return R[14];

// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
  BXWritePC(address);
aarch32/functions/registers/LookUpRIndex

// LookUpRIndex()
// =============

integer LookUpRIndex(integer n, bits(5) mode)
assert n >= 0 && n <= 14;
case n of // Select index by mode: usr fiq irq svc abt und hyp
    when 8 result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
    when 9 result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
    when 10 result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
    when 11 result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
    when 12 result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
    when 13 result = RBankSelect(mode, 13, 29, 13, 13, 13, 13, 13);
    when 14 result = RBankSelect(mode, 14, 30, 14, 14, 14, 14, 14);
    otherwise result = n;
return result;

aarch32/functions/registers/Monitor_mode_registers

bits(32) SP_mon;
bits(32) LR_mon;

aarch32/functions/registers/PC

// PC - non-assignment form
// =------------------------

bits(32) PC
return R[15]; // This includes the offset from AArch32 state

aarch32/functions/registers/PCStoreValue

// PCStoreValue()
// =============

bits(32) PCStoreValue()
// This function returns the PC value. On architecture versions before ARMv7, it
// is permitted to instead return PC+4, provided it does so consistently. It is
// used only to describe A32 instructions, so it returns the address of the current
// instruction plus 8 (normally) or 12 (when the alternative is permitted).
// return PC;

aarch32/functions/registers/Q

// Q[] - non-assignment form
// =======================

bits(128) Q[integer n]
assert n >= 0 && n <= 15;
return _V[n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
assert n >= 0 && n <= 15;
_V[n] = value;
return;
aarch32/functions/registers/Qin

// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];

aarch32/functions/registers/R

// R[] - assignment form
// ---------------------

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// -------------------------

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];

aarch32/functions/registers/RBankSelect

// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
    integer svc, integer abt, integer und, integer hyp)
    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode
    return result;

aarch32/functions/registers/Rmode

// Rmode[] - non-assignment form
// ----------------------------

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor
        if n == 13 then return SP_mon;
        elseif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
```c
return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// ==============

Rmode[integer n, bits(5) mode] = bits(32) value
assert n >= 0 & n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);
if mode == M32_Monitor then
  if n == 13 then SP_mon = value;
  elsif n == 14 then LR_mon = value;
  else _R[n]<31:0> = value;
else
  // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
  // register are unchanged or set to zero. This is also tested for on
  // exception entry, as this applies to all AArch32 registers.
  if !HighestELUsingAArch32() && ConstrainUnpredictableBool() then
    _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
  else
    _R[LookUpRIndex(n, mode)]<31:0> = value;
  end;
return;

aarch32/functions/registers/S

// S[] - non-assignment form
// ===============

bits(32) S[integer n]
assert n >= 0 & n <= 31;
base = (n MOD 4) * 32;
return _V[n DIV 4]<base+31:base>;

// S[] - assignment form
// ===============

S[integer n] = bits(32) value
assert n >= 0 & n <= 31;
base = (n MOD 4) * 32;
_V[n DIV 4]<base+31:base> = value;
return;

aarch32/functions/registers/SP

// SP - assignment form
// ===============

SP = bits(32) value
_R[13] = value;
return;

// SP - non-assignment form
// ===============

bits(32) SP
return R[13];

aarch32/functions/registers/_Dclone

array bits(64) _Dclone[0..31];
```
aarch32/functions/system/AArch32.ExceptionReturn

// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

// Attempts to change to an illegal mode or state will invoke the Illegal Execution state
// mechanism
SetPSTATEFromPSR(spsr);
ClearExclusiveLocal(ProcessorID());
EventRegisterSet();

// Align PC[1:0] according to the target instruction set state
if spsr<5> == '1' then // T32
  new_pc = Align(new_pc, 2);
else // A32
  new_pc = Align(new_pc, 4);
BranchTo(new_pc, BranchType_UNKNOWN);

aarch32/functions/system/AArch32.ITAdvance

// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
  if PSTATE.IT<2:0> == '000' then
    PSTATE.IT = '00000000';
  else
    PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
  return;

aarch32/functions/system/AArch32.SysRegRead

// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);

aarch32/functions/system/AArch32.SysRegRead64

// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);

aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
assert UsingAArch32();
assert (cp_num IN {14,15});
assert cp_num == UInt(instr<11:8>);

opc1 = UInt(instr<23:21>);
opc2 = UInt(instr<7:5>);
CRn = UInt(instr<19:16>);
CMn = UInt(instr<3:0>);

if cp_num == 14 && opc1 == 0 && CRn == 0 && CMn == 1 && opc2 == 0 then // DBGDSRInt
  return TRUE;

return FALSE;
// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);

// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);

// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.
AArch32.WriteMode(bits(5) mode)
  (valid,el) = ELFromM32(mode);
  assert valid;
  PSTATE.M = mode;
  PSTATE.EL = el;
  PSTATE.nRW = '1';
  PSTATE.SP = (if mode IN {M32_User,M32_System} then '0' else '1');
  return;

// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.
AArch32.WriteModeByInstr(bits(5) mode)
  (valid,el) = ELFromM32(mode);
  if UInt(el) > UInt(PSTATE.EL) then
    valid = FALSE;
  if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
    valid = FALSE;
  if !valid then
    PSTATE.IL = '1';
  else
    AArch32.WriteMode(mode);

// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
// case mode of
  when M32_Hyp
    valid = HaveEL(EL2);
  when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32.Undef, M32_System
    valid = TRUE;
  when M32_User
    valid = FALSE;
valid = TRUE;
otherwise
valid = FALSE;        // Passed an illegal mode value
return !valid;

aarch32/functions/system/BankedRegisterAccessValid
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.
BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)
  case SYSm of
    when '000xx', '00100'                          // R8_usr to R12_usr
      if mode != M32_FIQ then UNPREDICTABLE;
    when '00101'                                   // SP_usr
      if mode == M32_System then UNPREDICTABLE;
    when '00110'                                   // LR_usr
      if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
    when '010xx', '0110x', '01110'                  // R8_fiq to R12_fiq, SP_fiq, LR_fiq
      if mode == M32_FIQ then UNPREDICTABLE;
    when '1000x'                                   // LR_irq, SP_irq
      if mode == M32_IRQ then UNPREDICTABLE;
    when '1001x'                                   // LR_svc, SP_svc
      if mode == M32_Svc then UNPREDICTABLE;
    when '1010x'                                   // LR_abt, SP_abt
      if mode == M32_Abort then UNPREDICTABLE;
    when '1011x'                                   // LR_und, SP_und
      if mode == M32.Undef then UNPREDICTABLE;
    when '1110x'                                   // LR_mon, SP_mon
      if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
    when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
      if !HaveEL(EL2) || !(mode IN {M32_Monitor,M32_Hyp}) then UNPREDICTABLE;
    when '11111'                                   // SP_hyp, only from Monitor mode
      if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
    otherwise
      UNPREDICTABLE;
  return;

aarch32/functions/system/CPSRWriteByInstr
// CPSRWriteByInstr()
// ==================
CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
  privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0
  if bytemask<3> == '1' then
    PSTATE.<N,Z,C,V,Q> = value<31:27>;
    // Bits <26:24> are ignored
  if bytemask<2> == '1' then
    PSTATE.GE = value<19:16>;
    // Bits <15:10> are RES0
  if bytemask<1> == '1' then
    PSTATE.E = value<9>;                      // PSTATE.E is writable at EL0
    if privileged then
      PSTATE.A = value<8>;
  if bytemask<0> == '1' then

if privileged then
  PSTATE.<I,F> = value<7:6>;
  // Bit <5> is RES0
  // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
  AArch32.WriteModeByInstr(value<4:0>);

return;

aarch32/functions/system/ConditionPassed

// ConditionPassed()
// =========

boolean ConditionPassed()
return ConditionHolds(AArch32.CurrentCond());

aarch32/functions/system/CurrentCond

bits(4) AArch32.CurrentCond();

aarch32/functions/system/InITBlock

// InITBlock()
// ===========

boolean InITBlock()
if CurrentInstrSet() == InstrSet_T32 then
  return PSTATE.IT<3:0> != '0000';
else
  return FALSE;

aarch32/functions/system/LastInITBlock

// LastInITBlock()
// ===============

boolean LastInITBlock()
return (PSTATE.IT<3:0> == '1000');

aarch32/functions/system/SPSRWriteByInstr

// SPSRWriteByInstr()
// ================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

new_spsr = SPSR[];

if bytemask<3> == '1' then
  new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits

if bytemask<2> == '1' then

if bytemask<1> == '1' then
  new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

if bytemask<0> == '1' then
  new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits

SPSR[] = new_spsr;  // UNPREDICTABLE if User or System mode
return;
aarch32/functions/system/SPSRaccessValid

// SPSRaccessValid()
// ============
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
case SYSm of
  when '01110'                                                   // SPSR_fiq
    if mode == M32_FIQ   then UNPREDICTABLE;
  when '10000'                                                   // SPSR_irq
    if mode == M32_IRQ   then UNPREDICTABLE;
  when '10010'                                                   // SPSR_svc
    if mode == M32_Svc   then UNPREDICTABLE;
  when '10100'                                                   // SPSR_abt
    if mode == M32_Abt   then UNPREDICTABLE;
  when '10110'                                                   // SPSR_svc
    if mode == M32_Abt   then UNPREDICTABLE;
  when '11100'                                                   // SPSR_mon
    if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
  when '11110'                                                   // SPSR_hyp
    if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
otherwise
  UNPREDICTABLE;
return;

aarch32/functions/system/SelectInstrSet

// SelectInstrSet()
// ===============

SelectInstrSet(InstrSet iset)
  assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
  assert iset IN {InstrSet_A32, InstrSet_T32};
  PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
return;

aarch32/functions/v6simd/Sat

// Sat()
// ======

bits(N) Sat(integer i, integer N, boolean unsigned)
  result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
  return result;

aarch32/functions/v6simd/SignedSat

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
  (result, -) = SignedSatQ(i, N);
  return result;

aarch32/functions/v6simd/UnsignedSat

// UnsignedSat()
// ============
bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;

H1.2.4 aarch32/translation

aarch32/translation/attrs/AArch32.InstructionDevice

// AArch32.InstructionDevice()
// ================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress,
    bits(32) ipaddress, integer level, bits(4) domain,
    AccType acctype, boolean iswrite, boolean secondstage,
    boolean s2fslwalk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
            secondstage, s2fslwalk);
    else
        addrdesc.memattrs.type = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;

aarch32/translation/attrs/AArch32.ValidateAddressS1Off

// AArch32.ValidateAddressS1Off()
// ============================
// Called for getting default attributes when stage 1 validation is disabled.

MPURecord AArch32.ValidateAddressS1Off(bits(32) address, AccType acctype, boolean iswrite)

    MPURecord result;
    boolean secondstage = FALSE;
    result.br_enabled   = FALSE;
    default_cacheable   = HasS2Validation() && (HCR.DC == '1');

    if default_cacheable then
        // Use default cacheability settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;  // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Background memory map for data access
        result.addrdesc.memattrs = BackgroundMemoryAttr(address, secondstage);
    else
        // Background memory map for instruction fetch
        // Check for instruction fetch from execute-never memory address range
        if address<31> == '1' then
            result.perms.ap = bits(3) UNKNOWN;
            result.perms.xn = '1';

            if address<31> == '0' then
                result.perms.ab = bits(3) UNKNOWN;
                result.perms.xa = '1';

                if address<31> == '1' then
                    result.perms.ap = bits(3) UNKNOWN;
                    result.perms.xn = '1';
result.perms.pxn = '0';
result.addrdesc.paddress.physicaladdress = address;

// Instruction fetch from execute-never memory address range should be
// marked as permission fault and return.
result.addrdesc.fault = AArch32.PermissionFault(address,bits(4) UNKNOWN, integer UNKNOWN, 
acctype, iswrite, secondstage, FALSE);
return result;

else
// Instruction cacheability controlled by SCTLR/HSCTLR.I
if PSTATE.EL == EL2 then
  cacheable = HSCTLR.I == '1';
else
  cacheable = SCTLR.I == '1';
result.addrdesc.memattrs.type = MemType_Normal;
if cacheable then
  result.addrdesc.memattrs.inner.attrs    = MemAttr_WT;
  result.addrdesc.memattrs.inner.hints    = MemHint_RA;
  result.addrdesc.memattrs.shareable      = FALSE;
  result.addrdesc.memattrsoutershareable = FALSE;
else
  result.addrdesc.memattrs.inner.attrs    = MemAttr_NC;
  result.addrdesc.memattrs.inner.hints    = MemHint_No;
  result.addrdesc.memattrs.shareable      = TRUE;
  result.addrdesc.memattrsoutershareable = TRUE;

  // Use outer attributes as same as inner attributes
  // If no faults, populate rest of memattrs.
  result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
  result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

  // Update permission information
  result.perms.ap  = bits(3) UNKNOWN;
  result.perms.xn  = '0';
  result.perms.pxn = '0';
result.addrdesc.paddress.physicaladdress = address;
result.addrdesc.fault                    = AArch32.NoFault();
return result;

aarch32/translation/attrs/AArch32.ValidateAddressS2Off

// AArch32.ValidateAddressS2Off()
// ****************************
// Called for getting default attributes when stage 2 validation is disabled.

MPURecord AArch32.ValidateAddressS2Off(bits(32) address, AccType acctype, boolean iswrite)

  MPURecord result;
  boolean secondstage = TRUE;
  result.br_enabled   = FALSE;

  if acctype != AccType_IFETCH then
    // Background memory map for data access
    result.addrdesc.memattrs = BackgroundMemoryAttr(address, secondstage);
  else
    // Background memory map for instruction fetch
    // Check for instruction fetch from execute-never memory address range
    if address<31> == '1' then
      result.perms.ap  = bits(3) UNKNOWN;
      result.perms.xn  = '1';
      result.perms.pxn = '0';
      result.addrdesc.paddress.physicaladdress = address;
// Instruction fetch from execute-never memory address range should be
// marked as permission fault and return.
result.addrdesc.fault = AArch32.PermissionFault(address, bits(4) UNKNOWN, integer UNKNOWN,
  acctype, iswrite, secondstage, FALSE);
return result;

else
  // Stage2 validation happens for transactions from EL0/EL1 on PL2 MPU.
cacheable = HSCTRL.I == '1';
result.addrdesc.memattrs.type = MemType_Normal;
if cacheable then
  result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
  result.addrdesc.memattrs.innerhints = MemHint_RA;
  result.addrdesc.memattrs.shareable = FALSE;
  result.addrdesc.memattrs.outershareable = FALSE;
else
  result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
  result.addrdesc.memattrs.innerhints = MemHint_No;
  result.addrdesc.memattrs.shareable = TRUE;
  result.addrdesc.memattrs.outershareable = TRUE;

// Use outer attributes as same as inner attributes
// If no faults, populate rest of memattrs.
result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

// Update permission information
result.perms.ap = bits(3) UNKNOWN;
result.perms.xn = '0';
result.perms.pxn = '0';
result.addrdesc.paddress.physicaladdress = address;
result.addrdesc.fault = AArch32.NoFault();
return result;

aarch32/translation/checks/AArch32.CheckPermission

// AArch32.CheckPermission()
// =========================
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
  bits(4) domain, bit NS, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1ValidationRegime());

if PSTATE.EL != EL2 then
  wxn = SCTLR.WXN == '1';
  // v8R PMSA MPU follow Long-descriptor format
  priv_r = TRUE;
  priv_w = perms.ap<2> == '0';
  user_r = perms.ap<1> == '1';
  user_w = perms.ap<2:1> == '01';
  uwxn = SCTLR.UWXN == '1';
  user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
  priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
    (priv_w && wxn) || (user_w && uwxn));
  ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;
  if ispriv then
    (r, w, xn) = (priv_r, priv_w, priv_xn);
  else
    (r, w, xn) = (user_r, user_w, user_xn);
else
  // Access from EL2
  wxn = HSCTRL.WXN == '1';
r = TRUE;
w = perms.ap<2> == '0';

xn = perms.xn == '1' || (w && xn);

// Restriction on Secure instruction fetch
if acctype == AccType_IFETCH then
  fail = xn;
elsif iswrite then
  fail = !w;
  failedread = FALSE;
else
  fail = !r;
  failedread = TRUE;
endif

if fail then
  secondstage = FALSE;
  s2fs1walk = FALSE;
  ipaddress = bits(32) UNKNOWN;
  return AArch32.PermissionFault(ipaddress, domain, level, acctype, !failedread, secondstage, s2fs1walk);
else
  return AArch32.NoFault();
endif

aarch32/translation/checks/AArch32.CheckS2Permission

// AArch32.CheckS2Permission()
// ----------------------------------
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(32) ipaddress,
  integer level, AccType acctype, boolean iswrite,
  boolean s2fs1walk)

assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Validation();

r = perms.ap<1> == '1';
w = perms.ap<2> == '1';

xn = !r || perms.xn == '1';

// Stage 1 walk is checked as a read, regardless of the original type
if acctype == AccType_IFETCH && !s2fs1walk then
  fail = xn;
elsif iswrite && !s2fs1walk then
  fail = !w;
  failedread = FALSE;
else
  fail = !r;
  failedread = !iswrite;
endif

if fail then
  domain = bits(4) UNKNOWN;
  secondstage = TRUE;
  return AArch32.PermissionFault(ipaddress, domain, level, acctype, !failedread, secondstage, s2fs1walk);
else
  return AArch32.NoFault();
endif

aarch32/translation/debug/AArch32.CheckBreakpoint

// AArch32.CheckBreakpoint()
// ---------------------------
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32 translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction fetch. This is the simple sequential execution of the program.
FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1ValidationRegime());
assert size IN {2, 4};

match = FALSE;
mismatch = FALSE;

for i = 0 to UInt(DBGDIDR.BRPs)
    (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
    match = match || match_i;
    mismatch = mismatch || mismatch_i;
if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Breakpoint;
    Halt(reason);
elsif (match || mismatch) && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
iswrite = FALSE;
d debugmoe = DebugException_Breakpoint;
return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();

aarch32/translation/debug/AArch32.CheckDebug

// AArch32.CheckDebug()
// ================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

FaultRecord fault = AArch32.NoFault();
d_side = (acctype != AccType_IFETCH);
generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
halt = HaltOnBreakpointOrWatchpoint();
// Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
vector_catch_first = ConstrainUnpredictableBool();

if !d_side && vector_catch_first && generate_exception then
    fault = AArch32.CheckVectorCatch(vaddress, size);
if fault.type == Fault_None && (generate_exception || halt) then
    if d_side then
        fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
    else
        fault = AArch32.CheckBreakpoint(vaddress, size);
if fault.type == Fault_None && !d_side && !vector_catch_first && generate_exception then
    return AArch32.CheckVectorCatch(vaddress, size);
return fault;

aarch32/translation/debug/AArch32.CheckVectorCatch

// AArch32.CheckVectorCatch()
// ===============
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1ValidationRegime());
match = AArch32.VCRMatch(vaddress);
if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
    match = ConstrainUnpredictableBool();

if match && DBGSCExt.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
    accetype = AccType_IFETCH;
    iswrite = FALSE;
    debugmoe = DebugException_VectorCatch;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();

aarch32/translation/debug/AArch32.CheckWatchpoint

// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of “size” bytes at “address”.
FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
    boolean iswrite, integer size)
    assert ELUsingAArch32(S1ValidationRegime());
match = FALSE;
ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);
for i = 0 to UInt(DBGDIDR.WRPs)
    match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);
if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Watchpoint;
    Halt(reason);
elsif match && DBGSCExt.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
    debugmoe = DebugException_Watchpoint;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();

aarch32/translation/faults/AArch32.AccessFlagFault

// AArch32.AccessFlagFault()
// =========================
FaultRecord AArch32.AccessFlagFault(bits(32) ipaddress, bits(4) domain, integer level,
    AccType acctype, boolean iswrite, boolean secondstage,
    boolean s2fs1walk)
extflag = bit UNKNOWN;
d debugmoe = bits(4) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress, domain, level, acctype, iswrite,
    extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.AddressSizeFault

// AArch32.AddressSizeFault()
// ==========================
FaultRecord AArch32.AddressSizeFault(bits(32) ipaddress, bits(4) domain, integer level,
    AccType acctype, boolean iswrite, boolean secondstage,
    boolean s2fs1walk)
extflag = bit UNKNOWN;
d debugmoe = bits(4) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress, domain, level, acctype, iswrite,
    extflag, debugmoe, secondstage, s2fs1walk);
aarch32/translation/faults/AArch32.AlignmentFault

// AArch32.AlignmentFault()
//=-------------------------------------

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(32) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.AsynchExternalAbort

// AArch32.AsynchExternalAbort()
//=--------------------------------------
// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bit extflag)

type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
ipaddress = bits(32) UNKNOWN;
domain = bits(4) UNKNOWN;
level = integer UNKNOWN;
acctype = AccType_NORMAL;
iswrite = boolean UNKNOWN;
debugmoe = bits(4) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
    return AArch32.CreateFaultRecord(type, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.BackgroundFault

// AArch32.BackgroundFault()
//=----------------------------

FaultRecord AArch32.BackgroundFault(bits(32) ipaddress, bits(4) domain, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Background, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.DebugFault

// AArch32.DebugFault()
//=---------------------

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(32) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
```
return AArch32.CreateFaultRecord(Fault_Debug, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.DomainFault

// AArch32.DomainFault()
// =====================
FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

ipaddress = bits(32) UNKNOWN;
extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;

return AArch32.CreateFaultRecord(Fault_Domain, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.NoFault

// AArch32.NoFault()
// ===============
FaultRecord AArch32.NoFault()

ipaddress = bits(32) UNKNOWN;
domain = bits(4) UNKNOWN;
level = integer UNKNOWN;
acctype = AccType_NORMAL;
iswrite = boolean UNKNOWN;
extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;

return AArch32.CreateFaultRecord(Fault_None, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.PermissionFault

// AArch32.PermissionFault()
// =========================
FaultRecord AArch32.PermissionFault(bits(32) ipaddress, bits(4) domain, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)

extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_Permission, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/faults/AArch32.TranslationFault

// AArch32.TranslationFault()
// ==========================
FaultRecord AArch32.TranslationFault(bits(32) ipaddress, bits(4) domain, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)

extflag = bit UNKNOWN;
```
debugmoe = bits(4) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_Translation, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, secondstage, s2fs1walk);

aarch32/translation/validation/AArch32.AttrDecode

// AArch32.AttrDecode()
// ===============
// Converts the attribute fields, using the HMAIR/MAIR, to orthogonal attributes and hints.

MemoryAttributes AArch32.AttrDecode(bits(2) SH, bits(3) attr, AccType acctype, boolean secondstage)

MemoryAttributes memattrs;
if !secondstage && PSTATE.EL != EL2 then
    mair = MAIR1:MAIR0;
else
    mair = HMAIR1:HMAIR0;
index = 8 * UInt(attr);
attrfield = mair<index+7:index>;
if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
    (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
    // Reserved, maps to an allocated value
    (_, attrfield) = ConstrainUnpredictableBits();
if attrfield<7:4> == '0000' then            // Device
    memattrs.type = MemType_Device;
    case attrfield<3:0> of
        when '0000' memattrs.device = DeviceType_nGnRnE;
        when '0100' memattrs.device = DeviceType_nGnRE;
        when '1000' memattrs.device = DeviceType_nGRE;
        when '1100' memattrs.device = DeviceType_GRE;
        otherwise Unreachable();         // Reserved, handled above
    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.type  = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype, secondstage);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype, secondstage);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    else
        Unreachable();                       // Reserved, handled above
return MemAttrDefaults(memattrs);

aarch32/translation/validation/AArch32.CheckAddress

// AArch32.CheckAddress()
// =========
// Returns the result of address checks.

MPURecord AArch32.CheckAddress(bits(32) address, AccType acctype, boolean iswrite, boolean secondstage)
S1 = MPUCheck(address, acctype, iswrite, secondstage);
return S1;

aarch32/translation/validation/AArch32.FirstStageValidate

// AArch32.FirstStageValidate()
// ============
// Perform stage 1 validation for the memory access.
AddressDescriptor AArch32.FirstStageValidate(bits(32) address, AccType acctype, boolean iswrite, boolean wasaligned)

// Check whether stage 1 is enabled
if PSTATE_EL == EL2 then
  s1_enabled = HSCCTRL.M == '1';
elsif HaveEL(EL2) then
  tge = HCR.TGE;
dc = HCR.DC;
s1_enabled = tge == '0' && SCTLR.M == '1';
else
  dc = HCR.DC;
s1_enabled = && SCTLR.M == '1';

secondstage = FALSE;
if s1_enabled then
  // First stage enabled
  S1 = AArch32.CheckAddress(address, acctype, iswrite, secondstage);
  permissioncheck = TRUE;

  // Check whether to use background memory map or default cacheability
  if !s1_enabled || S1.br_enabled then
    S1 = AArch32.ValidateAddressS1Off(address, acctype, iswrite);
    permissioncheck = FALSE;

  // Check for unaligned data accesses to Device memory
  if !wasaligned && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device && acctype != AccType_IFETCH then
    S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
  if !IsFault(S1.addrdesc) && permissioncheck then
    S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, address, integer UNKNOWN, bits(4) UNKNOWN, '1', acctype, iswrite);

  // Check for instruction fetches from Device memory not marked as execute-never. If there has not been a Permission Fault then the memory is not marked execute-never.
  if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device && acctype == AccType_IFETCH) then
    S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, bits(32) UNKNOWN, address, integer UNKNOWN, bits(4) UNKNOWN, acctype, iswrite, secondstage, FALSE);
  return S1.addrdesc;

aarch32/translation/validation/AArch32.FullValidate

// AArch32.FullValidate()
// ======================
// Perform both stage 1 and stage 2 address validation for the current memory access regime.

AddressDescriptor AArch32.FullValidate(bits(32) address, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

// First Stage Validation
S1 = AArch32.FirstStageValidate(address, acctype, iswrite, wasaligned);

// Second Stage Validation
if !IsFault(S1) && HasS2Validation() then
  result = AArch32.SecondStageValidate(S1, address, acctype, iswrite, wasaligned, size);
else
  result = S1;
return result;
// AArch32.MPUCheck()
// ===============
// Returns the result of MPU checks on the address.

MPURecord MPUCheck(bits(32) address, AccType acctype, boolean iswrite, boolean secondstage)

MPURecord memrecord;
memrecord.br_enabled = FALSE;

if !secondstage && PSTATE.EL != EL2 then
  // Access controlled by EL1-controlled MPU
  pl2mpu = FALSE;
  br = SCTLR.BR == '1';
  num_regions = (UInt(MPUIR.REGION) - 1);
else
  // Access controlled by EL2-controlled MPU
  pl2mpu = TRUE;
  br = HSCTRL.BR == '1';
  num_regions = (UInt(HMPUIR.REGION) - 1);

regionMatched = FALSE;
for r = 0 to num_regions
  if pl2mpu then
    rbar = HPRBAR[r];
    rlar = HPRLAR[r];
  else
    rbar = PRBAR[r];
    rlar = PRLAR[r];
  // MPU region enable check
  if rlar.EN == '1' then
    // Checking for a matching MPU region
    if ((UInt(address) >= UInt(rbar.BASE : '00000')) &&
        (UInt(address) <= UInt(rlar.LIMIT : '11111'))) then
      // More than one region match will generate a translation fault
      if regionMatched then
        memrecord.addrdesc.fault = AArch32.TranslationFault(address,
                                                          bits(4) UNKNOWN, integer UNKNOWN,
                                                          acctype, iswrite, secondstage,
                                                          boolean UNKNOWN);
        memrecord.perms.ap = rbar.AP:'0';
        memrecord.perms.xn = rbar.XN;
        memrecord.perms.pxn = '0';
        memrecord.addrdesc.paddress.physicaladdress = address;
        SH = rbar.SH;
        attr = rlar.AttrIndx;
        memrecord.addrdesc.memattrs = AArch32.AttrDecode(SH, attr, acctype,
                                                        secondstage);
        memrecord.addrdesc.fault = AArch32.NoFault();
      else
        regionMatched = TRUE;
        memrecord.perms.ap = rbar.AP:'0';
        memrecord.perms.xn = rbar.XN;
        memrecord.perms.pxn = '0';
        memrecord.addrdesc.paddress.physicaladdress = address;
        SH = rbar.SH;
        attr = rlar.AttrIndx;
        memrecord.addrdesc.memattrs = AArch32.AttrDecode(SH, attr, acctype,
                                                        secondstage);
    // Check for single MPU region match
    if regionMatched then
      memrecord.addrdesc.fault = AArch32.NoFault();
  // In case of no match, check whether use of background memory region
  else br then
    memrecord.addrdesc.fault = AArch32.BackgroundFault(address, bits(4) UNKNOWN,
                                                         integer UNKNOWN, acctype, iswrite,
                                                         secondstage, boolean UNKNOWN);
// if background region is enabled, i.e. br = TRUE, check whether mpu background
// region matching is enabled for the current memory access regime.
// Check whether EL1-controlled MPU background region enabled
elsif !pl2mpu && PSTATE.EL == EL1 then
    memrecord.br_enabled = TRUE;
// Check whether EL2-controlled MPU background region enabled
elsif pl2mpu && PSTATE.EL==EL2 then
    memrecord.br_enabled = TRUE;
// else, raise translation fault
else
    memrecord.addrdesc.fault = AArch32.TranslationFault(address, bits(4) UNKNOWN,
             integer UNKNOWN, acctype, iswrite,
             secondstage, boolean UNKNOWN);

return memrecord;

aarch32/translation/validation/AArch32.SecondStageValidate

// AArch32.SecondStageValidate()
// =================================
// Perform a stage 2 validation for memory access.

AddressDescriptor AArch32.SecondStageValidate(AddressDescriptor S1, bits(32) address,
 AccType acctype, boolean iswrite, boolean wasaligned,
 integer size)

assert HasS2Validation();
s2_enabled = HCR.VM == '1' || HCR.DC == '1';
secondstage = TRUE;

// Second stage enabled
if s2_enabled then
    // Check whether stage2 MPU is enabled
    if HSCTLR.M == '1' then
        S2 = AArch32.CheckAddress(address, acctype, iswrite, secondstage);
        permissioncheck = TRUE;
    else
        // if stage2 MPU is disabled and HCR.VM == 1, then stage2 validation uses background
        // memory map
        S2 = AArch32.ValidateAddressS2Off(address, acctype, iswrite);
        permissioncheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
     acctype != AccType_IFETCH) then
        S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S2.addrdesc) && permissioncheck then
        S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, bits(32) UNKNOWN, address,
                                                 integer UNKNOWN, acctype,
                                                 iswrite, FALSE);

    // 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 (!IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
     acctype == AccType_IFETCH) then
        domain = bits(4) UNKNOWN;
        S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, bits(32) UNKNOWN, address,
                                                 integer UNKNOWN, acctype,
                                                 iswrite, secondstage, FALSE);
        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

return result;
aarch32/translation/validation/AArch32.ValidateAddress

// AArch32.ValidateAddress()
// ===============
// Main entry point for validating address for current memory access regime.

AddressDescriptor AArch32.ValidateAddress(bits(32) address, AccType acctype, boolean iswrite,
  boolean wasaligned, integer size)
result = AArch32.FullValidate(address, acctype, iswrite, wasaligned, size);
if !(acctype IN {AccType_IC}) && !IsFault(result) then
  result.fault = AArch32.CheckDebug(address, acctype, iswrite, size);
return result;

aarch32/translation/validation/BackgroundMemoryAttr

// BackgroundMemoryAttr()
// ================
// Called for getting background memory map attributes for a given address.

MemoryAttributes BackgroundMemoryAttr(bits(32) address, boolean secondstage)

  MemoryAttributes memattrs;
  memregion = UInt(address<31:28>);
  if PSTATE_EL != EL2 then {use SCTLR.C}
  else {use HSCTLR.C}
  // memory attributes not set here will be filled by MemAttrDefaults().
  if memregion >= UInt('1100') then
    // mpu region : 0xC0000000 - 0xFFFFFFFF
    memattrs.type   = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
  elsif memregion >= UInt('1000') then
    // mpu region : 0x80000000 - 0xBFFFFFFF
    memattrs.type   = MemType_Device;
    memattrs.device = DeviceType_nGnRE;
  elsif memregion >= UInt('0110') then
    // mpu region : 0x60000000 - 0x7FFFFFFF
    memattrs.type   = MemType_Normal;
    memattrs.inner.attrs    = MemAttr_NC;
    memattrs.inner.hints    = MemHint_No;
    memattrs.shareable      = TRUE;
    memattrs.outershareable = TRUE;
  elsif memregion >= UInt('0100') then
    // mpu region : 0x40000000 - 0x5FFFFFFF
    memattrs.type   = MemType_Normal;
    if cacheable then
      memattrs.inner.attrs    = MemAttr_WT;
      memattrs.inner.hints    = MemHint_RA;
      memattrs.shareable      = FALSE;
      memattrs.outershareable = FALSE;
    else
      memattrs.inner.attrs    = MemAttr_NC;
      memattrs.inner.hints    = MemHint_No;
      memattrs.shareable      = TRUE;
      memattrs.outershareable = TRUE;
    end
  else
    // mpu region : 0x00000000 - 0x3FFFFFFF
    memattrs.type   = MemType_Normal;
    if cacheable then
      memattrs.inner.attrs    = MemAttr_WT;
      memattrs.inner.hints    = MemHint_RA;
      memattrs.shareable      = FALSE;
      memattrs.outershareable = FALSE;
    else
      memattrs.inner.attrs    = MemAttr_NC;
      memattrs.inner.hints    = MemHint_No;
      memattrs.shareable      = TRUE;
      memattrs.outershareable = TRUE;
    end
  end
memattrs.inner.attrs = MemAttr_WB;
memattrs.inner.hints = MemHint_RWA;
memattrs.shareable = FALSE;
memattrsoutershareable = FALSE;

else
    memattrs.inner.attrs = MemAttr_NC;
    memattrs.inner.hints = MemHint_No;
    memattrs.shareable = TRUE;
    memattrs.outershareable = TRUE;

return memattrs;
H1.3 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. ARMv8-R does not support AArch64 state but for ease of correlating this manual with that on the ARMv8-A profile, this manual follows the structure of the ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

Functions listed in this section are identified only by a FunctionName, without an AArch64. or AArch32. prefix. 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:
- shared/debug.
- shared/exceptions on page H1-299.
- shared/functions on page H1-301.
- shared/translation on page H1-346.

H1.3.1 shared/debug

shared/debug/ClearStickyErrors/ClearStickyErrors

// ClearStickyErrors()
// ================
ClearStickyErrors()
EDSCR.TXU = '0'; // Clear TX underrun flag
EDSCR.RXO = '0'; // Clear RX overrun flag
if Halted() then // in Debug state
  EDSRC.ITO = '0'; // Clear ITR overrun flag
EDSCR.ERR = '0'; // Clear cumulative error flag
return;

shared/debug/DebugTarget/DebugTarget

// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
  return (if HDCR.TDE == '1' || HCR.TGE == '1' then EL2 else EL1);

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()
  return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();

shared/debug/FindWatchpoint/FindWatchpoint

// FindWatchpoint()
// ================

integer FindWatchpoint()
  address = FAR[];
  base = Align(address, ZVAGranuleSize());
limit = base + ZVAGranuleSize();
repeat
   for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
      if WatchpointByteMatch(i, address) then     // Candidate found
         return i;
      address = address + 1;
      if address == limit then address = base;        // Wrap round, as this must be a DC ZVA
   while address != FAR[];
   return -1;                                          // No candidate found (should not happen)

shared/debug/authentication/AllowExternalDebugAccess

// AllowExternalDebugAccess()
// =========================
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess()
   // The access may also be subject to OS lock, power-down, etc.
   return ExternalInvasiveDebugEnabled();

shared/debug/authentication/AllowExternalPMUAccess

// AllowExternalPMUAccess()
// =======================
// If the first return value is FALSE, then the access returns an error response. Otherwise, the second
// return value specifies a bit mask. In this bit mask, each zero bit is RAZ/WI in the access.
// If "indexed" == TRUE then this is a PMEVCNTR<n> or PMEVTYPER<n> register and "n" is the index of the
// event counter. If "mask" == TRUE then this is one of PMCNTENSET/CLR, PMINTENSET/CLR, or PMOVSSET/CLR.

(boolean,bits(32)) AllowExternalPMUAccess(boolean indexed, integer n, boolean mask)
   // The access may also be subject to OS lock, power-down, etc.
   if ExternalNoninvasiveDebugEnabled() then
      if HaveEL(EL2) && !ExternalHypNoninvasiveDebugEnabled() && HDCR.EPMAD == '1' then
         if mask then
            return (TRUE, '1':ZeroExtend(Ones(UInt(HDCR.HPMN)), 31));
         elsif indexed && n >= UInt(HDCR.HPMN) then
            return (TRUE, Zeros(32));
         return (TRUE, Ones());
      else
         return (FALSE, bits(32) UNKNOWN);
   shared/debug/authentication/Debug_authentication

signal DBGEN;
signal NIDEN;
signal HIDEN;
signal HNIDEN;

shared/debug/authentication/ExternalHypInvasiveDebugEnabled

// ExternalHypInvasiveDebugEnabled()
// =================================
// In the recommended interface, ExternalHypInvasiveDebugEnabled returns the state of the
// (DBGEN AND HIDEN) signal.

boolean ExternalHypInvasiveDebugEnabled()
   return ExternalInvasiveDebugEnabled() && HIDEN == HIGH;

shared/debug/authentication/ExternalHypNoninvasiveDebugEnabled

// ExternalHypNoninvasiveDebugEnabled()
// ====================================

boolean ExternalHypNoninvasiveDebugEnabled()
// In the recommended interface, ExternalHypNoninvasiveDebugEnabled returns the state of the
// (DBGEN OR NIDEN) AND (HIDEN OR HNIDEN) signal.
return ExternalNoninvasiveDebugEnabled() && (HIDEN == HIGH || HNIDEN == HIGH);

shared/debug/authentication/ExternalInvasiveDebugEnabled

// ExternalInvasiveDebugEnabled()
// ---------------------------------
boolean ExternalInvasiveDebugEnabled()
// In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN
// signal.
return DBGEN == HIGH;

shared/debug/authentication/ExternalNoninvasiveDebugAllowed

// ExternalNoninvasiveDebugAllowed()
// ---------------------------------
boolean ExternalNoninvasiveDebugAllowed()
// Return TRUE if Trace and Sample-based profiling are allowed
return ExternalNoninvasiveDebugEnabled();

shared/debug/authentication/ExternalNoninvasiveDebugEnabled

// ExternalNoninvasiveDebugEnabled()
// ---------------------------------
boolean ExternalNoninvasiveDebugEnabled()
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
// OR NIDEN) signal.
return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;

shared/debug/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_RSV03,
                          CrossTriggerOut_TraceExtIn0, CrossTriggerOut_TraceExtIn1,
                          CrossTriggerOut_TraceExtIn2, CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt, CrossTriggerIn_PMUOverflow,
                            CrossTriggerIn_RSV02, CrossTriggerIn_RSV03,
                            CrossTriggerIn_TraceExtOut0, CrossTriggerIn_TraceExtOut1,
                            CrossTriggerIn_TraceExtOut2, CrossTriggerIn_TraceExtOut3};

shared/debug/dccanditr/CheckForDCCInterrupts

// CheckForDCCInterrupts()
// ------------------------
CheckForDCCInterrupts()
// ------------------------
comrx = (EDSCR.RXfull == '1');
comtx = (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 & DBGDCINT.RX == '1') ||
            (commtx & DBGDCINT.TX == '1'));
SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

return;

shared/debug/dccanditr(DBGDTRRX_EL0

// DBGDTRRX_EL0[] (external write)
// ============================================================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then
    // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return;
if EDSCR.ERR == '1' then return;

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return;
if EDSCR.RXfull == '1' || (Halted() & EDSCR.RXO == '1') then
    EDSCR.RXO = '1';  EDSCR.ERR = '1';
    // Overrun condition: ignore write
    return;

EDSCR.RXfull = '1';
DTRRX = value;

if Halted() && EDSCR.RA == '1' then
    EDSCR.ITE = '0';
    // See comments in EDITR[] (external write)
    ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
    ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
    R[1] = bits(32) UNKNOWN;

    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(32) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';
        // See comments in EDITR[] (external write)
        return;

    // DBGDTRRX_EL0[] (external read)
    // ============================================================

    bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;

shared/debug/dccanditr(DBGDTRTX_EL0

// DBGDTRTX_EL0[] (external read)
// ============================================================
// Called on reads of debug register 0x080.
bits(32) DBGDTRTX_EL0[boolean memory_mapped]

if EDPSR<6:5,0> != '001' then
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN;
end if

if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN
end if

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
endif

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
    return value;
endif

if underrun then
    EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
    return value;                                   // Return UNKNOWN
endif

EDSCR.TXfull = '0';

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)
    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
        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';
    endif
    R[1] = bits(32) UNKNOWN;
endif

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 DBGDTR_EL0<[Xt]> 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 "signal slave-generated error";
return;
if EDSCR.ERR == '1' then return;  // Error flag set: ignore write
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.ITE = '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 then handle_el = PSTATE_EL;
      elseif HaveEL(EL2) && !IsSecure() && HCR.TGE == '1' then UNDEFINED;
      else handle_el = EL1;
    when EL2
      if !HaveEL(EL2) || EDSCR.HDD == '1' then UNDEFINED;
      else handle_el = EL2;
      otherwise
        UNDEFINED;

      if ELUsingAArch32(handle_el) then
        assert UsingAArch32(); // Cannot move from AArch64 to AArch32
        case handle_el of
          when EL1
            AArch32.WriteMode(M32_Svc);
          when EL2
            AArch32.WriteMode(M32_Hyp);
          if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
          else
            LR = bits(32) UNKNOWN;
            SPSR[] = bits(32) UNKNOWN;
          if handle_el == EL2 then
            PSTATE.E = HSCTLR.EE;
          else
            PSTATE.E = SCTLR.EE;
          DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
          UpdateEDSCRFields(); // Update EDSCR PE state flags.
          return;

shared/debug/halting/DRPSInstruction

// DRPSInstruction()
// ================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

  SynchronizeContext();

  SetPSTATEFromPSR(SPSR[]);

  // PSTATE.(N,Z,C,V,Q,GE,SS,D,A,I,F) are not observable and ignored in Debug state, so
  // behave as if UNKNOWN.
  if UsingAArch32() then
    PSTATE.(N,Z,C,V,Q,GE,SS,D,A,I,F) = bits(13) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.T = '00000000';  PSTATE.T = '1'; // PSTATE.T is RES0
    DLR = bits(32) UNKNOWN;  DSPSR = 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 = ZeroExtend(DLR);
spsr = DSPSR;
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
SetPSTATEFromPSR(spsr);                            // Can update privileged bits, even at EL0.

if UsingAArch32() then
  if ConstrainUnpredictableBool() then new_pc<0> = '0';
  BranchTo(new_pc,<31:0>, BranchType_UNKNOWN);  // AArch32 branch
else
  // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
  if spsr<4> == '1' && ConstrainUnpredictableBool() then
    new_pc<63:32> = Zeros();
  BranchTo(new_pc, BranchType_DBGEXIT);          // A type of branch that is never predicted

(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');        // Atomically signal restarted
UpdateEDSCRFields();                               // Stop signalling PE state.
DisableITRAndResumeInstructionPrefetch();

return;
shared/debug/halting/Halt

// Halt()
//=

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt
    DLR = ThisInstrAddr();
    DSPSR = GetPSRFromPSTATE();
    
    EDSCR.ITE = '1';  EDSCR.ITO = '0';
    if HaveEL(EL2) && (PSTATE.EL == EL2 || HCR.TGE == '1') then
        EDSCR.HDD = '0'; // If entered in Hyp or with TGE set, allow debug
    elsif HaveEL(EL2) then
        EDSCR.HDD = (if ExternalHypInvasiveDebugEnabled() then '0' else '1');
    else
        assert EDSCR.HDD == '1'; // Otherwise EDSCR.HDD is RES1
        EDSCR.MA = '0';
    // PSTATE.{A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    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' && DBGOSLSR.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 HaveEL(EL2) && (PSTATE.EL == EL2 || HCR.TGE == '1') then
        return ExternalHypInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();

**shared/debug/halting/Restarting**

```java
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';  // Restarting
```

**shared/debug/halting/StopInstructionPrefetchAndEnableITR**

```java
StopInstructionPrefetchAndEnableITR();
```

**shared/debug/halting/UpdateEDSCRFields**

```java
// UpdateEDSCRFields()
// ===============

// Update EDSCR PE state fields

UpdateEDSCRFields()
    if !Halted() then
        EDSCR.EL = '00';
    else
        EDSCR.EL = PSTATE.EL;
    return;
```

**shared/debug/haltingevents/CheckExceptionCatch**

```java
// CheckExceptionCatch()
// ================

// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch()
    if HaltingAllowed() && EDECCR<UInt(PSTATE.EL) + base> == '1' then
        Halt(DebugHalt_ExceptionCatch);
```

**shared/debug/haltingevents/CheckHaltingStep**

```java
// CheckHaltingStep()
// =============

// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elseif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

**shared/debug/haltingevents/CheckOSUnlockCatch**

```java
// CheckOSUnlockCatch()
// ===============

// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';
```
shared/debug/haltingevents/CheckPendingOSUnlockCatch

// CheckPendingOSUnlockCatch()
// ========================================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
if HaltingAllowed() && EDESR.OSUC == '1' then
    Halt(DebugHalt_OSUnlockCatch);

shared/debug/haltingevents/CheckPendingResetCatch

// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
if HaltingAllowed() && EDESR.RC == '1' then
    Halt(DebugHalt_ResetCatch);

shared/debug/haltingevents/CheckResetCatch

// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
if EDECR.RCE == '1' then
    EDESR.RC = '1';
    // If halting is allowed then halt immediately
    if HaltingAllowed() then Halt(DebugHalt_ResetCatch);

shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
if HaltingAllowed() && EDSCR.TDA == '1' && DBGOSLSR.OSLK == '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 == EL2 then
        advance = syscall || ExternalHypInvasiveDebugEnabled();
    else
        advance = TRUE;
    if advance then EDESR.SS = '1';
return;

shared/debug/interrupts/InterruptID

enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
    InterruptID_COMMRX, InterruptID_COMMTX};

shared/debug/interrupts/SetInterruptRequestLevel

// Set a level-sensitive interrupt to the specified level. 
SetInterruptRequestLevel(InterruptID id, signal level);

shared/debug/samplebasedprofiling/CreatePCSample

// CreatePCSample()
// ===============

CreatePCSample()
// In a simple sequential execution of the program, CreatePCSample is executed each time the PE
// executes an instruction that can be sampled. An implementation is not constrained such that
// reads of EDPCSR return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.contextidr = CONTEXTIDR;
    if HaveEL(EL2) && !IsSecure() then
        pc_sample.vmid = VSCTLR.VMID;
        return;

shared/debug/samplebasedprofiling/EDPCSR

// EDPCSR[] (read)
// ===============

bits(32) EDPCSR[boolean memory_mapped]

    if EDPSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";


return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        EDCIDS = pc_sample.contextidr;
        EDVIDSR.VMID = (if HaveEL(EL2) & pc_sample.el IN {EL1,EL0} then pc_sample.vmid else Zeros(8));
        EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
    else
        sample = Ones(32);
    if update then
        EDCIDS = bits(32) UNKNOWN;
        EDVIDSR = (bits(4) UNKNOWN):Zeros(20):(bits(8) UNKNOWN);
return sample;

shared/debug/samplebasedprofiling/PCSample

type PCSample is (
    boolean valid,
    bits(32) pc,
    bits(2) el,
    bits(32) contextidr,
    bits(8) vmid
)

PCSample pc_sample;

H1.3.2 shared/exceptions

shared/exceptions/exceptions/ConditionSyndrome

// ConditionSyndrome()
// ===============
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

bits(5) syndrome;

if UsingAArch32() then
    cond = AArch32.CurrentCond();
    if PSTATE.T == '0' then // A32
        syndrome<4> = '1';
        // A conditional A32 instruction that is known to pass its condition code check
        // can be presented either with COND set to 0x0, the value for unconditional, or
        // the COND value held in the instruction.
        if ConditionHolds(cond) & ConstrainUnpredictableBool() then
            syndrome<3:0> = '1111';
        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';
        else
            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_CP15RRRTTrap, // Trapped AArch32 MCRR or MRRC access to CP15
    Exception_CP14RTTrap,   // Trapped AArch32 MCR or MRC access to CP14
    Exception_CP14DTTrap,   // Trapped AArch32 LDC or STC access to CP14
    Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap,     // Trapped access to SIMD or FP ID register
    Exception_CP14RTRTTrap, // Trapped CP14 as CP14 from AArch32
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_MonitorCall,   // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCAlignment,   // Misaligned PC
    Exception_DataAbort,     // Data Abort
    Exception_SPAAlignment,  // Misaligned SP
    Exception_FPTrapException, // 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_IRQHandler,  // IRQ interrupt
    Exception_FIQ}         // FIQ interrupt

shared/exceptions/exceptions/ExceptionRecord

type ExceptionRecord is (Exception type, // Exception class
    bits(25) syndrome, // Syndrome record
    bits(64) vaddress, // Virtual fault address
    boolean ipavalid, // Physical fault address is valid
    bits(48) ipaddress) // Physical fault address for second stage faults

shared/exceptions/exceptions/ExceptionSyndrome

// ExceptionSyndrome()
// ===============
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception type)
{
    ExceptionRecord r;
    r.type = type;
    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.ipaddress = Zeros();
    return r;
}
shared/functions

shared/functions/aborts/EncodeLDFSC

// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault type, integer level)

    bits(6) result;
    case type of
        when Fault_AddressSize result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal result = '010000';
        when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity result = '011000';
        when Fault_SyncParityOnWalk result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsycnParity result = '01001';
        when Fault_AsycnExternal result = '010001';
        when Fault_AccessFlag result = '100001';
        when Fault_Debug result = '100010';
        when Fault_TLBConflict result = '110000';
        when Fault_Lockdown result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive result = '110101';  // IMPLEMENTATION DEFINED
        otherwise Unreachable();

    return result;

shared/functions/aborts/FaultSyndrome

// FaultSyndrome()
// ===============
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode or an Exception Level using AArch64.

bits(25) FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(25) iss = Zeros();
    if d_side then
        if IsSecondStage(fault) & & !fault.s2fs1walk then iss<24:16> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        if fault.s2fs1walk then '1' else '0';
        else
            iss<5:0> = EncodeLDFSC(fault.type, fault.level);
    return iss;

shared/functions/aborts/IPAValid

// IPAValid()
// ===========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault_None;

    if fault.s2fs1walk then
return fault.type IN {Fault_AccessFlag, Fault_Permission, Fault_Translation, Fault_AddressSize};
elsif fault.secondstage then
  return fault.type IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
else
  return FALSE;

shared/functions/aborts/IsDebugException

// IsDebugException()
// ===============

boolean IsDebugException(FaultRecord fault)
  assert fault.type != Fault_None;
  return fault.type == Fault_Debug;

shared/functions/aborts/IsExternalAbort

// IsExternalAbort()
// ================
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault type)
  assert type != Fault_None;
  return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_AsyncExternal, Fault_AsyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk});

// IsExternalAbort()
// ================

boolean IsExternalAbort(FaultRecord fault)
  return IsExternalAbort(fault.type);

shared/functions/aborts/IsFault

// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
  return addrdesc.fault.type != Fault_None;

shared/functions/aborts/IsSErrorInterrupt

// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE otherwise.

boolean IsSErrorInterrupt(Fault type)
  assert type != Fault_None;
  return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
  return IsSErrorInterrupt(fault.type);
shared/functions/aborts/IsSecondStage

// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.secondstage;

shared/functions/aborts/LSInstructionSyndrome

bits(11) LSInstructionSyndrome();

shared/functions/common/ASR

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

shared/functions/common/ASR_C

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

shared/functions/common/Abs

// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;

shared/functions/common/Align

// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);
Bits(N) Align(bits(N) x, integer y)
return Align(UInt(x), y)<N-1:0>;

Shared/Functions/Common/BitCount

// BitCount()
// ===========

Integer BitCount(bits(N) x)
integer result = 0;
for i = 0 to N-1
  if x<i> == '1' then
    result = result + 1;
return result;

Shared/Functions/Common/CountLeadingSignBits

// CountLeadingSignBits()
// ================

Integer CountLeadingSignBits(bits(N) x)
return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);

Shared/Functions/Common/CountLeadingZeroBits

// CountLeadingZeroBits()
// ================

Integer CountLeadingZeroBits(bits(N) x)
return N - 1 - HighestSetBit(x);

Shared/Functions/Common/Elem

// Elem[] - non-assignment form
// =============

Bits(size) Elem[bits(N) vector, integer e, integer size]
assert e >= 0 & (e+1)*size <= N;
return vector<e+size*size-1 : e*size>;

// Elem[] - non-assignment form
// =============

Bits(size) Elem[bits(N) vector, integer e]
return Elem[vector, e, size];

// Elem[] - assignment form
// =============

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
assert e >= 0 & (e+1)*size <= N;
vector<(e+1)*size-1:e+size> = value;
return;

// Elem[] - assignment form
// =============

Elem[bits(N) &vector, integer e] = bits(size) value
Elem[vector, e, size] = value;
return;
shared/functions/common/Extend

// Extend()
// =========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
  return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// =========

bits(N) Extend(bits(M) x, boolean unsigned)
  return Extend(x, N, unsigned);

shared/functions/common/HighestSetBit

// HighestSetBit()
// ================

integer HighestSetBit(bits(N) x)
  for i = N-1 downto 0
    if x<i> == '1' then return i;
  return -1;

shared/functions/common/Int

// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
  result = if unsigned then UInt(x) else SInt(x);
  return result;

shared/functions/common/IsOnes

// IsOnes()
// =======

boolean IsOnes(bits(N) x)
  return x == Ones(N);

shared/functions/common/IsZero

// IsZero()
// ========

boolean IsZero(bits(N) x)
  return x == Zeros(N);

shared/functions/common/IsZeroBit

// IsZeroBit()
// ==========

bit IsZeroBit(bits(N) x)
  return if IsZero(x) then '1' else '0';

shared/functions/common/LSL

// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
assert shift >= 0;
if shift == 0 then
    result = x;
else
    (result, -) = LSL_C(x, shift);
return result;

shared/functions/common/LSL_C

// LSL_C()
// ======

<bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

shared/functions/common/LSR

// LSR()
// =====

<bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

shared/functions/common/LSR_C

// LSR_C()
// ======

<bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

shared/functions/common/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/NOT
    bits(N) NOT(bits(N) x);

shared/functions/common/Ones
    // Ones()
    // ======
    bits(N) Ones(integer N)
        return Replicate('1',N);
    // Ones()
    // ======
    bits(N) Ones()
        return Ones(N);

shared/functions/common/ROR
    // ROR()
    // ======
    bits(N) ROR(bits(N) x, integer shift)
        assert shift >= 0;
        if shift == 0 then
            result = x;
        else
            (result, -) = ROR_C(x, shift);
        return result;

shared/functions/common/ROR_C
    // ROR_C()
    // ======
    (bits(N), bit) ROR_C(bits(N) x, integer shift)
        assert shift != 0;
        m = shift MOD N;
        result = LSR(x,m) OR LSL(x,N-m);
        carry_out = result<N-1>;
        return (result, carry_out);
shared/functions/common/Replicate

// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
assert N MOD M == 0;
return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);

shared/functions/common/RoundDown

integer RoundDown(real x);

shared/functions/common/RoundTowardsZero

// RoundTowardsZero()
// =============

integer RoundTowardsZero(real x)
return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);

shared/functions/common/RoundUp

integer RoundUp(real x);

shared/functions/common/SInt

// SInt()
// =====

integer SInt(bits(N) x)
result = 0;
for i = 0 to N-1
  if x[i] == '1' then result = result + 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<N-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 boolean IMPLEMENTATION_DEFINED "Have CRC extension";

shared/functions/crc/Poly32Mod2

// Poly32Mod2()
// =============
// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
if data<i> == '1' then
  data<i-1:0> = data<i-1:0> EOR poly:Zeros(i-32);
return data<31:0>;

shared/functions/crypto/AESInvMixColumns
bits(128) AESInvMixColumns(bits (128) op);

shared/functions/crypto/AESInvShiftRows
bits(128) AESInvShiftRows(bits(128) op);

shared/functions/crypto/AESInvSubBytes
bits(128) AESInvSubBytes(bits(128) op);

shared/functions/crypto/AESMixColumns
bits(128) AESMixColumns(bits (128) op);

shared/functions/crypto/AESShiftRows
bits(128) AESShiftRows(bits(128) op);

shared/functions/crypto/AESSubBytes
bits(128) AESSubBytes(bits(128) op);

shared/functions/crypto/HaveCryptoExt
boolean HaveCryptoExt();

shared/functions/crypto/ROL
// ROL()
// =====
bits(N) ROL(bits(N) x, integer shift)
  assert shift >= 0 && shift <= N;
  if (shift == 0) then
    return x;
  return ROR(x, N-shift);

shared/functions/crypto/SHA256hash
// SHA256hash()
// ============
bits(128) SHA256hash (bits (128) X, bits(128) Y, bits(128) W, boolean part1)
bits(32) chs, maj, t;
for e = 0 to 3
  chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
  maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
  t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
  X<127:96> = t + X<127:96>;
  Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
  <Y, X> = ROL(Y : X, 32);
return (if part1 then X else Y);
shared/functions/crypto/SHAchoose

// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
return (((y EOR z) AND x) EOR z);

shared/functions/crypto/SHAhashSIGMA0

// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);

shared/functions/crypto/SHAhashSIGMA1

// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);

shared/functions/crypto/SHAmajority

// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
return ((x AND y) OR ((x OR y) AND z));

shared/functions/crypto/SHAparity

// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
return (x EOR y EOR z);

shared/functions/exclusive/ClearExclusiveByAddress

// ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/ClearExclusiveLocal

// ClearExclusiveLocal(integer processorid);

shared/functions/exclusive/ClearExclusiveMonitors

// ClearExclusiveMonitors()
// ==============

// Clear the local Exclusive Monitor for the executing PE.
ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
shared/functions/exclusive/ExclusiveMonitorsStatus

// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();

shared/functions/exclusive/IsExclusiveGlobal

// Return TRUE if the global Exclusive Monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/IsExclusiveLocal

// Return TRUE if the local Exclusive Monitor for processorid includes all of
// the physical address region of size bytes starting at 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 Exclusive Monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/MarkExclusiveLocal

// Record the physical address region of size bytes starting at paddress in
// the local Exclusive Monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/ProcessorID

// Return the ID of the currently executing PE.
integer ProcessorID();

shared/functions/float/fixedtofp/FixedToFP

// FixedToFP()
// ===========
// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and Rounding.
bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {32,64};
assert M IN {32,64};
bits(N) result;
assert fbits >= 0;
assert rounding != FPRounding_ODD;

// Correct signed-ness
int_operand = Int(op, unsigned);

// Scale by fractional bits and generate a real value
real_operand = Real(int_operand) / 2.0^fbits;
if real_operand == 0.0 then
  result = FPZero('0');
else
result = FPRound(real_operand, fpcr, rounding);
return result;

shared/functions/float/fpabs/FPAbs

// FPAbs()
// ========

bits(N) FPAbs(bits(N) op)
assert N IN {32,64};
return '0' : op[N-2:0];

shared/functions/float/fpadd/FPAdd

// FPAdd()
// =========

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPTYPE_INFINITY); inf2 = (type2 == FPTYPE_INFINITY);
  zero1 = (type1 == FPTYPE_ZERO); zero2 = (type2 == FPTYPE_ZERO);
  if inf1 && inf2 && sign1 == NOT(sign2) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  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;
else
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()
  if value1 == value2 then
    result = '0100';
  elseif value1 < value2 then
    result = '3000';
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 {32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
    result = FALSE;
    if type1==FPType_SNaN || type2==FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 == value2);
return result;

shared/functions/float/fpcomparege/FPCompareGE

// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
    result = FALSE;
    FPProcessException(FPExc_InvalidOp, fpcr);
else // All non-NaN cases can be evaluated on the values produced by 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 {32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
    result = FALSE;
    FPProcessException(FPExc_InvalidOp, fpcr);
else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 > value2);
return result;

shared/functions/float/fpconvert/FPConvert

// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.

bits(M) FPConvert(bits(N) op, FPCRType fpocr, FPRounding rounding)
assert M IN (16, 32, 64);
assert N IN (16, 32, 64);
bits(M) result;

// Unpack floating-point operand optionally with flush-to-zero.
(type, sign, value) = FPUnpack(op, fpcr);
alt_hp = (M == 16) && (fpcr.AHP == '1');
if type == FType_SNaN || type == FType_QNaN then
    if alt_hp then
        result = FPZero(sign);
    elsif fpcr.DN == '1' then
        result = FPDefaultNaN();
    else
        result = FPConvertNaN(op);
    endif
elsif type == FType_SNaN || alt_hp then
    FPProcessException(FPExcp_InvalidOp, fpcr);
else
    result = FPConvertNaN(op);
endif
result = FPInfinity(sign);
result = FPZero(sign);
else
    result = FPRound(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' return FPRounding_TIEAWAY; // A
    when '01' return FPRounding_TIEEVEN; // N
    when '10' return FPRounding_POSINF;  // P
    when '11' return FPRounding_NEGINF;  // M

shared/functions/float/fpdecoderounding/FPDecodeRounding

// FPDecodeRounding()
// ==============

// Decode floating-point rounding mode and common AArch64 encoding.
FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
    when '00' return FPRounding_TIEEVEN; // N
    when '01' return FPRounding_POSINF;  // P
    when '10' return FPRounding_NEGINF;  // M
    when '11' return FPRounding_ZERO;    // Z

shared/functions/float/fpdefaultnan/FPDefaultNaN

// FPDefaultNaN()
// ===============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;

shared/functions/float/fpdiv/FPDiv

// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
result = FPInfinity(sign1 EOR sign2);
if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
elsif zero1 || inf2 then
    result = FPZero(sign1 EOR sign2);
else
    result = FPRound(value1/value2, fpcr);
return result;

shared/functions/float/fpexc/FPExc

enumeration FPExc
    {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
    FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

shared/functions/float/fpinfinity/FPInfinity

    // FPInfinity()
    // ============

    bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;

shared/functions/float/fpmax/FPMax

    // FPMax()
    // =========

    bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
        return result;

shared/functions/float/fpmaxnormal/FPMaxNormal

    // FPMaxNormal()
    // ===========

    bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
shared/functions/float/fpmaxnum/FPMaxNum

    // FPMaxNum()
    // =========

    bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');
    end

    return FPMax(op1, op2, fpcr);

shared/functions/float/fpmin/FPMin

    // FPMin()
    // =======

    bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        end
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
        end
        return result;
    end

shared/functions/float/fpminnum/FPMinNum

    // FPMinNum()
    // ==========

    bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');
    end

    return FPMin(op1, op2, fpcr);
shared/functions/float/fpmul/FPMul

// FPMul()
// ========

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN (32, 64);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if (inf1 && zero2) || (zero1 && inf2) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif inf1 || inf2 then
    result = FPInfinity(sign1 EOR sign2);
  elsif zero1 || zero2 then
    result = FPZero(sign1 EOR sign2);
  else
    result = FPRound(value1*value2, fpcr);
return result;

shared/functions/float/fmuladd/FPMulAdd

// FPMulAdd()
// =========

// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN (32, 64);
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
(done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);
if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
  result = FPDefaultNaN();
  FPProcessException(FPExc_InvalidOp, fpcr);
if !done then
  infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
  // Determine sign and type product will have if it does not cause an Invalid
  // Operation.
  signP = sign1 EOR sign2;
  infP = inf1 || inf2;
  zeroP = zero1 || zero2;
  // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
  // additions of opposite-signed infinities.
  if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  // Other cases involving infinities produce an infinity of the same sign.
  elsif (infA && signA == '0') || (infP && signP == '0') then
    result = FPInfinity('0');
  elsif (infA && signA == '1') || (infP && signP == '1') then
    result = FPInfinity('1');
// Cases where the result is exactly zero and its sign is not determined by the
// rounding mode are additions of same-signed zeros.
else if zeroA && zeroP && signA == signP then
    result = FPZero(signA);

// Otherwise calculate numerical result and round it.
else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr);
    return result;

shared/functions/float/fpmulx/FPMulX

// FPMulX()
// =========
bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = 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 {32,64};
    return NOT(op<N-1>) : op<N-2:0>;

shared/functions/float/fponepointfive/FPOnePointFive

// FPOnePointFive()
// ================
bits(N) FPOnePointFive(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
shared/functions/float/fpprocessexception/FPProcessException

// FPProcessException()
// ===============
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
// Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;

shared/functions/float/fpprocessnan/FPProcessNaN

// FPProcessNaN()
// ================

bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRType fpcr)
assert N IN {32,64};
assert type IN {FPType_QNaN, FPType_SNaN};
topfrac = if N == 32 then 22 else 51;
result = op;
if type == FPType_SNaN then
    result<topfrac> = '1';
FPProcessException(FPExc_InvalidOp, fpcr);
if fpcr.DN == '1' then  // DefaultNaN requested
    result = FPDefaultNaN();
return result;

shared/functions/float/fpprocessnans/FPProcessNaNs

// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
bits(N) op1, bits(N) op2,
FPCRType fpcr)
assert N IN {32,64};
if type1 == FPType_SNaN then
done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
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 {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 {32, 64};
(type,sign,value) = FPUnpack(operand, fpcr);
if type == FPType_SNaN || type == FPType_QNaN then
    result = FPProcessNaN(type, operand, fpcr);
elsif type == FPType_Infinity then
    result = FPZero(sign);
elsif type == FPType_Zero then
    result = FPInfinity(sign);
    FPProcessException(FPExc_DivideByZero, fpcr);
elsif (N == 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 == '-');
            result = FPInfinity(sign);
            FPProcessException(FPExc_DivideByZero, fpcr);
    end
else
    result = FPZero(sign);
return (done, result);
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 == 32 && Abs(value) >= 2.0^126)
|| (N == 64 && Abs(value) >= 2.0^1022)) then

// Result flushed to zero of correct sign
result = FPZero(sign);
if UsingAArch32() then
    FPSCR.UFC = '1';
else
    FPSR.UFC = '1';
else

// Scale to a double-precision value in the range 0.5 <= x < 1.0, and
// calculate result exponent. Scaled value has copied sign bit,
// exponent = 1022 = double-precision biased version of -1,
// fraction = original fraction extended with zeros.
if N == 32 then
    fraction = operand<22:0> : Zeros(29);
    exp = UInt(operand<30:23>);
else // N == 64
    fraction = operand<51:0>;
    exp = UInt(operand<62:52>);

if exp == 0 then

    if fraction<51> == 0 then
        exp = -1;
        fraction = fraction<49:0>:'00';
    else
        fraction = fraction<50:0>:'0';

    scaled = '0' : '01111111110' : fraction<51:44> : Zeros(44);

if N == 32 then
    result_exp = 253 - exp;     // In range 253-254 = -1 to 253+1 = 254
else // N == 64
    result_exp = 2045 - exp;    // In range 2045-2046 = -1 to 2045+1 = 2046

// Call C function to get reciprocal estimate of scaled value.
// Input is rounded down to a multiple of 1/512.
estimate = recip_estimate(scaled);

// Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
// Convert to scaled single-precision result with copied sign bit and high-order
// fraction bits, and exponent calculated above.

fraction = estimate<51:0>;
if result_exp == 0 then
    fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
    fraction = '01' : fraction<51:2>;
    result_exp = 0;
if N == 32 then
    result = sign : result_exp<5:0> : fraction<51:29>;
else // N == 64
    result = sign : result_exp<5:0> : fraction<51:0>;

return result;
bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
  assert N IN {32,64};
  bits(N) result;
  integer esize = if N == 32 then 8 else 11;
  bits(esize) exp;
  bits(esize) max_exp;
  bits(N-esize-1) frac = Zeros();
  if N == 32 then
    exp = op<23+esize-1:23>;
  else
    exp = op<52+esize-1:52>;
  max_exp = Ones(esize) - 1;
  (type,sign,value) = FPUnpack(op, fpcr);
  if type == FPType_SNaN || type == FPType_QNaN then
    result = FPProcessNaN(type, op, fpcr);
  else
    if IsZero(exp) then // Zero and denormals
      result = sign:max_exp:frac;
    else // Infinities and normals
      result = sign:NOT(exp):frac;
  return result;

shared/functions/float/fpround/FPRound

// FPRound()
// =========
// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
  assert N IN {16,32,64};
  assert op != 0.0;
  assert rounding != FPRounding_TIEAWAY;
  bits(N) result;

  // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
  if N == 16 then
    minimum_exp = -14;  E = 5;  F = 10;
  elsif N == 32 then
    minimum_exp = -126;  E = 8;  F = 23;
  else  // N == 64
    minimum_exp = -1022;  E = 11;  F = 52;

  // Split value into sign, unrounded mantissa and exponent.
  if op < 0.0 then
    sign = '1';  mantissa = -op;
  else
    sign = '0';  mantissa = op;
  exponent = 0;
  while mantissa < 1.0 do
    mantissa = mantissa * 2.0;  exponent = exponent - 1;
  while mantissa >= 2.0 do
    mantissa = mantissa / 2.0;  exponent = exponent + 1;

  // Deal with flush-to-zero.
  if fpcr.FZ == '1' && N != 16 && exponent < minimum_exp then
    // Flush-to-zero never generates a trapped exception
    if UsingAArch32() then
      FPSCR.UFC = '1';
    else
      FPSR.UFC = '1';
    return FPZero(sign);
// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
based_exp = Max(exponent - minimum_exp + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^\text{(minimum_exp - exponent)};

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = \text{RoundDown(mantissa \times 2.0^F)};  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa \times 2.0^F - \text{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(FPEExc_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 \text{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 \text{FPInfinity(sign)} else \text{FPMaxNormal(sign)};
    FPProcessException(FPEExc_Overflow, fpcr);
    error = 1.0;  // Ensure that an Inexact exception occurs
  else
    result = sign : biased_exp<\text{N-F-2:0}> : int_mant<F-1:0>;
else  // Alternative half precision
  if biased_exp >= 2^E then
    result = sign : \text{Ones(N-1)};
    FPProcessException(FPEExc_InvalidOp, fpcr);
    error = 0.0;  // Ensure that an Inexact exception does not occur
  else
    result = sign : biased_exp<\text{N-F-2:0}> : int_mant<F-1:0>;

// Deal with Inexact exception.
if error != 0.0 then
  FPProcessException(FPEExc_Inexact, fpcr);

return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
return FPRound(op, fpcr, FPRoundingMode(fpcr));
shared/functions/float/fprounding/FPRounding

enumeration FPRounding {FPRounding_TIEEVEN, FPRounding_POSINF, FPRounding_NEGINF, FPRounding_ZERO, FPRounding_TIEAWAY, FPRounding_ODD};

shared/functions/float/fproundingmode/FPRoundingMode

// FPRoundingMode()
// ================

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
return FPDecodeRounding(fpcr.RMode);

shared/functions/float/fproundint/FPRoundInt

// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode Rounding.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
assert rounding != FPRounding_ODD;
assert N IN {32,64};

// Unpack using FPCR to determine if subnormals are flushed-to-zero
(type,sign,value) = FPUnpack(op, fpcr);
if type == FPType_SNaN || type == FPType_QNaN then
result = FPProcessNaN(type, op, fpcr);
elsif type == FPType_Infinity then
result = FPInfinity(sign);
elsif type == FPType_Zero then
result = FPZero(sign);
else
// extract integer component
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment
case rounding of
when FPRounding_TIEEVEN
round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
when FPRounding_POSINF
round_up = (error != 0.0);
when FPRounding_NEGINF
round_up = FALSE;
when FPRounding_ZERO
round_up = (error != 0.0 && int_result < 0);
when FPRounding_TIEAWAY
round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
if round_up then int_result = int_result + 1;

// Convert integer value into an equivalent real value
real_result = Real(int_result);

// Re-encode as a floating-point value, result is always exact
if real_result == 0.0 then
result = FPZero(sign);
else
result = FPRound(real_result, fpcr, FPRounding_ZERO);
// Generate inexact exceptions
if error != 0.0 && exact then
    FPProcessException(FPExc_Inexact, fpcr);
return result;

shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

// FPRSqrtEstimate()
// ===============

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
assert N IN {32, 64};
(type,sign,value) = FPUnpack(operand, fpcr);
if type == FPType_SNan || type == FPType_QNaN then
    result = FPProcessNaN(type, operand, fpcr);
elsif type == FPType_Zero then
    result = FPInfinity(sign);
    FPProcessException(FPExc_DivideByZero, fpcr);
elsif sign == '1' then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
elsif type == FPType_Infinity then
    result = FPZero('0');
else
    // Scale to a double-precision value in the range 0.25 <= x < 1.0, with the
    // evenness or oddness of the exponent unchanged, and calculate result exponent.
    // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
    // biased version of -1 or -2, fraction = original fraction extended with zeros.
    if N == 32 then
        fraction = operand<22:0> : Zeros(29);
        exp = UInt(operand<30:23>);
    else // N == 64
        fraction = operand<51:0>;
        exp = UInt(operand<62:52>);
    if exp == 0 then
        while fraction<51> == 0 do
            fraction = fraction<50:0> : '0';
            exp = exp - 1;
        fraction = fraction<50:0> : '0';
    if exp<0> == '0' then
        scaled = '0' : '01111111101' : fraction<51:44> : Zeros(44);
    else
        scaled = '0' : '01111111101' : fraction<51:44> : Zeros(44);
    if N == 32 then
        result_exp = (380 - exp) DIV 2;
    else // N == 64
        result_exp = (3068 - exp) DIV 2;
    // Call C function to get reciprocal estimate of scaled value.
    estimate = recip_sqrt_estimate(scaled);
    // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
    // Convert to scaled single-precision result with copied sign bit and high-order
    // fraction bits, and exponent calculated above.
    if N == 32 then
        result = '0' : result_exp<31:25> : estimate<31:29>;
    else // N == 64
        result = '0' : result_exp<63:4> : estimate<31:0>;
    return result;
shared/functions/float/fpsqrt/FPSqrt

// FPSqrt()
// =========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
assert N IN {32,64};
(type,sign,value) = FPUnpack(op, fpcr);
if type == FPPType_SNaN || type == FPPType_QNaN then
  result = FPProcessNaN(type, op, fpcr);
elsif type == FPPType_Zero then
  result = FPZero(sign);
elsif type == FPPType_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 {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 == FPPType_Infinity);
  inf2 = (type2 == FPPType_Infinity);
  zero1 = (type1 == FPPType_Zero);
  zero2 = (type2 == FPPType_Zero);
  if inf1 && inf2 && sign1 == sign2 then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
    result = FPInfinity('0');
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPInfinity('1');
  elsif zero1 && zero2 && sign1 == NOT(sign2) then
    result = FPZero(sign1);
  else
    result_value = value1 - value2;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding);
      return result;
  end

shared/functions/float/fpthree/FPThree

// FPThree()
// =========

bits(N) FPThree(bit sign)
assert N IN {32,64};
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - E - 1;
exp = '1':Zeros(E-1);
frac = '1':Zeros(F-1);
return sign : exp : frac;

shared/functions/float/fptofixed/FPToFixed

// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {32,64};
assert M IN {32,64};
assert fbits >= 0;
assert rounding != FPRounding_ODD;
// Unpack using fpcr to determine if subnormals are flushed-to-zero
(type,sign,value) = FPUnpack(op, fpcr);
// If NaN, set cumulative flag or take exception
if type == FPType_SNaN || type == FPType_QNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
// Scale by fractional bits and produce integer rounded towards minus-infinity
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);
// Determine whether supplied rounding mode requires an increment
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_result < 0) == '1'));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 && int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
if round_up then int_result = int_result + 1;
// Generate saturated result and exceptions
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPExc_InvalidOp, fpcr);
elsif error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
return result;

shared/functions/float/fptwo/FPTwo

// FPTwo()
// ========

bits(N) FPTwo(bit sign)
assert N IN {32,64};
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - E - 1;
exp = '1':Zeros(E-1);
frac = Zeros(F);
return sign : exp : frac;
shared/functions/float/fptype/FPType

enumeration FPType {FPType_Nonzero, FPType_Zero, FPType_Infinity, FPType_QNaN, FPType_SNaN};

shared/functions/float/fpunpack/FPUnpack

// FPUnpack()
// =========
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
assert N IN {16,32,64};

if N == 16 then
    sign   = fpval<15>;
    exp16  = fpval<14:10>;
    frac16 = fpval<9:0>;
    if IsZero(exp16) then
        // Produce zero if value is zero
        if IsZero(frac16) then
            type = FPType_Zero;  value = 0.0;
        else
            type = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        else
            if IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
                if IsZero(frac16) then
                    type = FPType_Infinity;  value = 2.0^1000000;
                else
                    type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                    value = 0.0;
                else
                    type = FPType_Nonzero;
                    value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);
                endif
            else // N == 32 then
                sign   = fpval<31>;
                exp32  = fpval<30:23>;
                frac32 = fpval<22:0>;
                if IsZero(exp32) then
                    // Produce zero if value is zero or flush-to-zero is selected.
                    if IsZero(frac32) || fpcr.FZ == '1' then
                        type = FPType_Zero;  value = 0.0;
                    else
                        type = FPType_Nonzero; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
                    else
                        if IsOnes(exp32) then
                            if IsZero(frac32) then
                                type = FPType_Infinity;  value = 2.0^1000000;
                            else
                                type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                                value = 0.0;
                            else
                                type = FPType_Nonzero;
                                value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
                            endif
                        else // N == 64 then
                            // ...
                        endif
                    endif
                endif
            endif
        endif
    endif
endif
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) || fpccr.FZ == '1' then
    type = FPType_Zero;  value = 0.0;
  if !IsZero(frac64) then   // Denormalized input flushed to zero
    FPProcessException(FPExc_InputDenorm, fpccr);
  else
    type = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
elsif IsOnes(exp64) then
  if IsZero(frac64) then
    type = FPType_Infinity;  value = 2.0^1000000;
  else
    type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
    value = 0.0;
  else
    type = FPType_Nonzero;
    value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
  if sign == '1' then value = -value;
return (type, sign, value);

shared/functions/float/fpzero/FPZero

// FPZero()
// ========

bits(N) FPZero(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - E - 1;
exp  = Zeros(E);
frac = Zeros(F);
return sign : exp : frac;

shared/functions/float/vfpexpandimm/VFPExpandImm

// VFPExpandImm()
// ================

bits(N) VFPExpandImm(bits(8) imm8)
assert N IN {32,64};
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - E - 1;
sign = imm8<7>;
exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
frac = imm8<3:0>:Zeros(F-4);
return sign : exp : frac;

shared/functions/integer/AddWithCarry

// AddWithCarry()
// ===============

// Integer addition with carry input, returning result and NZCV flags
(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);  
integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
bit n = IsZero(result) then '1' else '0';
bit c = if UInt(result) == unsigned_sum then '0' else '1';
bit v = if SInt(result) == signed_sum then '0' else '1';
return (result, n:z:c:v);
shared/functions/memory/AccType

enumeration AccType {AccType_NORMAL, AccType_VEC,       // Normal loads and stores
AccType_STREAM, AccType_VECSTREAM,          // Streaming loads and stores
AccType_ATOMIC,                             // Atomic loads and stores
AccType_ORDERED,                            // Load-Acquire and Store-Release
AccType_UNPRIV,                             // Load and store unprivileged
AccType_IFETCH,                             // Instruction fetch
AccType_PTW,                                // Page table walk
// Other operations
AccType_DC,                                 // Data cache maintenance
AccType_IC,                                 // Instruction cache maintenance
AccType_AT};                                 // Address translation

shared/functions/memory/AddressDescriptor

type AddressDescriptor is ( FaultRecord     fault,      // fault.type indicates whether the address is valid
                          MemoryAttributes memattrs,
                          FullAddress      paddress,
                          bits(32)         vaddress
)

shared/functions/memory/Allocation

constant bits(2) MemHint_No = '00';     // No allocate
constant bits(2) MemHint_WA = '01';     // Write-allocate, Read-no-allocate
constant bits(2) MemHint_RA = '10';     // Read-allocate, Write-no-allocate
constant bits(2) MemHint_RWA = '11';    // Read-allocate and Write-allocate

shared/functions/memory/BigEndian

// BigEndian()
// =========

boolean BigEndian()

  boolean bigend;

  bigend = (PSTATE.E != '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/DataFullBarrier

DataFullBarrier();
shared/functions/memory/DataMemoryBarrier

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

shared/functions/memory/DataSynchronizationBarrier

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);

shared/functions/memory/DeviceType

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};

shared/functions/memory/Fault


shared/functions/memory/FaultRecord

type FaultRecord is (Fault type, // Fault Status
AccType acctype, // Type of access that faulted
bits(32) ipaddress,
boolean s2fs1walk, // Is on a Stage 1 page table walk
boolean write, // TRUE for a write, FALSE for a read
integer level, // For translation, access flag and permission faults
bit extflag, // IMPLEMENTATION DEFINED syndrome for external aborts
boolean secondstage, // Is a Stage 2 abort
bits(4) domain, // Domain number, AArch32 only
bits(4) debugmoe) // Debug method of entry, from AArch32 only

shared/functions/memory/FullAddress

type FullAddress is (
    bits(32) physicaladdress
)

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 specified address into one or more caches as
// indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint stream.
// Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a synchronous
// abort due to alignment or translation faults and the like. Its only effect on software visible
// state should be on caches and TLBs associated with address, which must be accessible by reads,
// writes or execution as defined in the translation regime of the current Exception level.
// It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);

shared/functions/memory/MBReqDomain
enumeration MBReqDomain {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                         MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

shared/functions/memory/MBReqTypes
enumeration MBReqTypes {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};

shared/functions/memory/MPURecord
type MPURecord is (Permissions perms,
                  AddressDescriptor addrdesc,
                  boolean br_enabled)

shared/functions/memory/MemAttrHints
type MemAttrHints is (bits(2) attrs, // The possible encodings for each attributes field are as below
                       bits(2) hints, // The possible encodings for the hints are below
                       boolean transient)

shared/functions/memory/MemType
enumeration MemType {MemType_Normal, MemType_Device};

shared/functions/memory/MemoryAttributes
type MemoryAttributes is (MemType type,
                          DeviceType device, // For Device memory types
                          MemAttrHints inner, // Inner hints and attributes
                          MemAttrHints outer, // Outer hints and attributes
                          boolean shareable,
                          boolean outershareable)

shared/functions/memory/Permissions
type Permissions is (bits(3) ap, // Access permission bits
                    bit xn, // Execute-never bit
                    bit pxn // Privileged execute-never bit)

shared/functions/memory/PrefetchHint
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
shared/functions/memory/TLBRecord

type TLBRecord is (  
  Permissions     perms,  
  bit            nG,       // '0' = Global, '1' = not Global  
  bits(4)        domain,  // AArch32 only  
  boolean        contiguous, // Contiguous bit from page table  
  integer        level,  // In AArch32 Short-descriptor format, indicates Section/Page  
  integer        blocksize, // Describes size of memory translated in KBytes  
  AddressDescriptor addrdesc
)

shared/functions/memory/_Mem

// These two _Mem[] accessors are the hardware operations which perform  
// single-copy atomic, aligned, little-endian memory accesses of size  
// bytes from/to the underlying physical memory array of bytes.  
//  
// The functions address the array using desc.PADDRESS which supplies:  
//  
// * A 48-bit physical address  
// * A single NS bit to select between Secure and Non-secure parts of  
//   the array.  
//  
// The acctype parameter describes the access type: normal, exclusive,  
// ordered, streaming, etc.  
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccType acctype];  
_Mem[AddressDescriptor desc, integer size, AccType acctype] = bits(8*size) value;

shared/functions/registers/BranchTo

// BranchTo()  
// ===========  
// Set program counter to a new address, which may include a tag in the top eight bits,  
// with a branch reason hint for possible use by hardware fetching the next instruction.  
BranchTo(bits(N) target, BranchType branch_type)  
  Hint_Branch(branch_type);  
  if N == 32 then  
    assert UsingAArch32();  
    _PC = ZeroExtend(target);  
  return;

shared/functions/registers/BranchToAddr

// BranchToAddr()  
// ===============  
// Set program counter to a new address, which does not include a tag in the top eight bits,  
// with a branch reason hint for possible use by hardware fetching the next instruction.  
BranchToAddr(bits(N) target, BranchType branch_type)  
  Hint_Branch(branch_type);  
  if N == 32 then  
    assert UsingAArch32();  
    _PC = ZeroExtend(target);  
  return;

shared/functions/registers/BranchType

enumeration BranchType {BranchType_CALL, BranchType_ERET, BranchType_DBGEXIT,  
  BranchType_RET, BranchType_JMP, BranchType_EXCEPTION,  
  BranchType_UNKNOWN};
shared/functions/registers/Hint_Branch

// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);

shared/functions/registers/NextInstrAddr

// Return address of the next instruction.
bits(N) NextInstrAddr();

shared/functions/registers/ResetExternalDebugRegisters

// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);

shared/functions/registers/ThisInstrAddr

// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 & UsingAArch32());
    return _PC<N-1:0>;

shared/functions/registers/_PC

bits(64) _PC;

shared/functions/registers/_R

array bits(64) _R[0..30];

shared/functions/registers/_V

array bits(128) _V[0..31];

shared/functions/sysregisters/SPSR

// SPSR[] - non-assignment form
// ===============

bits(32) SPSR[]
    bits(32) result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ     result = SPSR_fiq;
            when M32_IRQ      result = SPSR_irq;
            when M32_Svc      result = SPSR_svc;
            when M32_Abort    result = SPSR_abt;
            when M32_Hyp      result = SPSR_hyp;
            when M32_Undef    result = SPSR_und;
            otherwise        Unreachable();
        return result;

    // SPSR[] - assignment form
    // ===============

    SPSR[] = bits(32) value
    if UsingAArch32() then
        case PSTATE.M of
when M32_FIQ      SPSR_fiq = value;
when M32_IRQ      SPSR_irq = value;
when M32_Svc      SPSR_svc = value;
when M32_Abort    SPSR_abt = value;
when M32_Hyp      SPSR_hyp = value;
when M32_Undef    SPSR_und = value;
otherwise         Unreachable();

return;

shared/functions/system/ArchVersion

enumeration ArchVersion {
    ARMv8p0,
};

shared/functions/system/ClearEventRegister

ClearEventRegister();

shared/functions/system/ConditionHolds

// ConditionHolds()
// =============

// Return TRUE if COND currently holds

boolean ConditionHolds(bits(4) cond)
// Evaluate base condition.
    case cond<3:1> of
      when '000' result = (PSTATE.Z == '1'); // EQ or NE
      when '001' result = (PSTATE.C == '1'); // CS or CC
      when '010' result = (PSTATE.N == '1'); // MI or PL
      when '011' result = (PSTATE.V == '1'); // VS or VC
      when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
      when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
      when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
      when '111' result = TRUE; // AL

      // Condition flag values in the set '111x' indicate always true
      // Otherwise, invert condition if necessary.
      if cond<0> == '1' && cond != '1111' then
        result = !result;

      return result;

shared/functions/system/CurrentInstrSet

// CurrentInstrSet()
// ==============

InstrSet CurrentInstrSet()

if UsingAArch32() then
    result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
    // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
else
    result = InstrSet_A64;
return result;
shared/functions/system/CurrentPL

// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);

shared/functions/system/EL0

constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';

shared/functions/system/ELFromM32

// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //   and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    case mode of
      when M32_Monitor
        el = EL3;
      when M32_Hyp
        el = EL2;
      when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
        el = EL1;
      when M32_User
        el = EL0;
      otherwise
        valid = FALSE;           // Passed an illegal mode value
        if !valid then el = bits(2) UNKNOWN;
        return (valid, el);

shared/functions/system/ELFromSPSR

// ELFromSPSR()
// =============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    return ELFromM32(spsr<4:0>);

shared/functions/system/ELUsingAArch32

// ELUsingAArch32()
// ===============

boolean ELUsingAArch32(bits(2) el)
    if !HaveEL(el) then
        return FALSE;                    // The Exception level is not implemented
    else
        return TRUE;
shared/functions/system/EndOfInstruction

// Terminate processing of the current instruction.
EndOfInstruction();

shared/functions/system/EventRegisterSet

// Set the local event register in this PE.
EventRegisterSet();

shared/functions/system/EventRegistered

boolean EventRegistered();

shared/functions/system/GetPSRFromPSTATE

// GetPSRFromPSTATE()
// ================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
bits(32) spsr = Zeros();
spsr<31:28> = PSTATE.<N,Z,C,V>;
spsr<21> = PSTATE.SS;
spsr<20> = PSTATE.IL;
if PSTATE.nRW == '1' then // AArch32 state
  spsr<27> = PSTATE.Q;
spsr<26:25> = PSTATE.IT<1:0>;
spsr<19:16> = PSTATE.GE;
spsr<15:10> = PSTATE.IT<7:2>;
spsr<9> = PSTATE.E;
spsr<8:6> = PSTATE.<A,I,F>;
  assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
spsr<4:0> = PSTATE.M;
else // AArch64 state
  spsr<9:6> = PSTATE.<D,A,I,F>;
spsr<4> = PSTATE.nRW;
spsr<3:2> = PSTATE.EI;
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/HaveAnyAArch32

// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
  return boolean IMPLEMENTATION_DEFINED;
shared/functions/system/HaveEL

// HaveEL()
// =========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE;                             // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;

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_Yield

Hint_Yield();

shared/functions/system/IllegalExceptionReturn

// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for return:
    // * 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;

    // Check for return to EL1 in Non-secure state when HCR.TGE is set
    if HaveEL(EL2) && target == EL1 && HCR.TGE == '1' then return TRUE;
    return FALSE;

shared/functions/system/InstrSet

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
shared/functions/system/InstructionSynchronizationBarrier

InstructionSynchronizationBarrier();

shared/functions/system/InterruptPending

boolean InterruptPending();

shared/functions/system/IsSecure

// IsSecure()
// ========

boolean IsSecure()
    return FALSE;

shared/functions/system/Mode_Bits

constant bits(5) M32_User = '10000';
constant bits(5) M32_FIQ  = '10001';
constant bits(5) M32_IRQ  = '10010';
constant bits(5) M32_Svc  = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort = '10111';
constant bits(5) M32_Hyp  = '11010';
constant bits(5) M32_Undef = '11011';
constant bits(5) M32_System = '11111';

shared/functions/system/PLOfEL

// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
case el of
  when EL3 return if HighestELUsingAArch32() then PL1 else PL3;
  when EL2 return PL2;
  when EL1 return PL1;
  when EL0 return PL0;

shared/functions/system/PSTATE

ProcState PSTATE;

shared/functions/system/PrivilegeLevel

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};

shared/functions/system/ProcState

type ProcState is

  bits (1) N,      // Negative condition flag
  bits (1) Z,      // Zero condition flag
  bits (1) C,      // Carry condition flag
  bits (1) V,      // oVerflow condition flag
  bits (1) D,      // Debug mask bit
  [AArch64 only]
  bits (1) A,      // SError interrupt mask bit
  bits (1) I,      // IRQ mask bit
  bits (1) F,      // FIQ mask bit
  bits (1) S,      // 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
shared/functions/system/RestoredITBits

// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
   it = spsr<15:10,26:25>;
   // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
   // to zero or copied from the SPSR.
   if PSTATE.IL == '1'
      if ConstrainUnpredictableBool() then return '00000000';
      else return it;
   // The IT bits are forced to zero when they are set to a reserved value.
   if !IsZero(it<7:4>) && !IsZero(it<3:0>) then
      return '00000000';
   // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
   // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
   itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
   if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
      return '00000000';
   else
      return it;

shared/functions/system/SCRType
type SCRType;

shared/functions/system/SendEvent

// Signal an event to all PEs.
SendEvent();

shared/functions/system/SetPSTATEFromPSR

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value
SetPSTATEFromPSR(bits(32) spsr)
   SynchronizeContext();
   if IllegalExceptionReturn(spsr) then
      PSTATE.IL = '1';
   else
      // State that is reinstated only on a legal exception return
      PSTATE.IL = spsr<20>;
      // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
      // the T bit is set to zero or copied from SPSR.
      if PSTATE.IL == '1'
         if ConstrainUnpredictableBool() then spsr<5> = '0';

// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
   it = spsr<15:10,26:25>;
   // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
   // to zero or copied from the SPSR.
   if PSTATE.IL == '1'
      if ConstrainUnpredictableBool() then return '00000000';
      else return it;
   // The IT bits are forced to zero when they are set to a reserved value.
   if !IsZero(it<7:4>) && !IsZero(it<3:0>) then
      return '00000000';
   // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
   // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
   itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
   if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
      return '00000000';
   else
      return it;

shared/functions/system/SCRType
type SCRType;

shared/functions/system/SendEvent

// Signal an event to all PEs.
SendEvent();

shared/functions/system/SetPSTATEFromPSR

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value
SetPSTATEFromPSR(bits(32) spsr)
   SynchronizeContext();
   if IllegalExceptionReturn(spsr) then
      PSTATE.IL = '1';
   else
      // State that is reinstated only on a legal exception return
      PSTATE.IL = spsr<20>;
      // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
      // the T bit is set to zero or copied from SPSR.
      if PSTATE.IL == '1'
         if ConstrainUnpredictableBool() then spsr<5> = '0';
// State that is reinstated regardless of illegal exception return
PSTATE.<N,Z,C,V> = spsr<31:28>;
PSTATE.Q = spsr<27>;
PSTATE.IT = RestoredITBits(spsr);
PSTATE.GE = spsr<19:16>;
PSTATE.E = spsr<9>;
PSTATE.<A,I,F> = spsr<8:6>;
PSTATE.T = spsr<5>;
return;

shared/functions/system/SynchronizeContext
SynchronizeContext();

shared/functions/system/ThisInstr
bits(32) ThisInstr();

shared/functions/system/ThisInstrLength
integer ThisInstrLength();

shared/functions/system/Unreachable
Unreachable()
    assert FALSE;

shared/functions/system/UsingAArch32
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;

shared/functions/system/WaitForEvent
WaitForEvent();

shared/functions/system/WaitForInterrupt
WaitForInterrupt();

shared/functions/unpredictable/ConstrainUnpredictable
// Return the appropriate Constraint result to control the caller’s behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)
Constraint ConstrainUnpredictable();
shared/functions/unpredictable/ConstrainUnpredictableBits

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONstrained.
(Constraint,bits(width)) ConstrainUnpredictableBits();

shared/functions/unpredictable/ConstrainUnpredictableBool

// ConstrainUnpredictableBool()
// ============================
// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

boolean ConstrainUnpredictableBool()

    c = ConstrainUnpredictable();
    assert c IN [Constraint_TRUE, Constraint_FALSE];
    return (c == Constraint_TRUE);

shared/functions/unpredictable/ConstrainUnpredictableInteger

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.
(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high);

shared/functions/unpredictable/Constraint

enumeration Constraint    {// General:
    Constraint_NONE, Constraint_UNKNOWN,
    Constraint_UNDEF, Constraint_NOP,
    Constraint_TRUE, Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND, Constraint_COND, Constraint_ADDITIONAL_DECODE,
    // Load-store:
    Constraint_WBSUPPRESS, Constraint_FAULT,
    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLCHECK};

shared/functions/vector/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(imm8:Zeros(8):imm8, 4);
        when '101' imm64 = Replicate(imm8:Zeros(8), 4);
        when '110' imm64 = Replicate(imm8:Zeros(8), 4);
        when '111' imm64 = Replicate(imm8:Zeros(24), 2);
    end;

    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>, 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);
  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(32) UnsignedRSqrtEstimate(bits(32) operand)
  if operand<31:30> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
    result = Ones(32);
  else
    // Generate double-precision value = operand * 2^(-32). This has zero sign bit, with:
// exponent = 1022 or 1021 = double-precision representation of 2^(-1) or 2^(-2)
// fraction taken from operand, excluding its most significant one or two bits.
if operand<31> == '1' then
    dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
else // operand<31:30> == '01'
    dp_operand = '0 01111111101' : operand<29:0> : Zeros(22);

// Call C function to get reciprocal estimate of scaled value.
estimate = recip_sqrt_estimate(dp_operand);

// Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
// Multiply by 2^31 and convert to an unsigned integer - this just involves
// concatenating the implicit units bit with the top 31 fraction bits.
result = '1' : estimate<51:21>;
return result;

shared/functions/vector/UnsignedRecipEstimate

// UnsignedRecipEstimate()
// ================
bits(32) UnsignedRecipEstimate(bits(32) operand)

if operand<31> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones(32);
else
    // Generate double-precision value = operand + 2^(-32). This has zero sign bit, with:
    // exponent = 1022 = double-precision representation of 2^(-1)
    // fraction taken from operand, excluding its most significant bit.
    dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);

    // Call C function to get reciprocal estimate of scaled value.
    estimate = recip_estimate(dp_operand);

    // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
    // Multiply by 2^31 and convert to an unsigned integer - this just involves
    // concatenating the implicit units bit with the top 31 fraction bits.
    result = '1' : estimate<51:21>;
return result;

shared/functions/vector/UnsignedSatQ

// UnsignedSatQ()
// =============
(bits(N), boolean) UnsignedSatQ(integer i, integer N)

if i > 2^N - 1 then
    result = 2^N - 1;  saturated = TRUE;
elif i < 0 then
    result = 0;  saturated = TRUE;
else
    result = i;  saturated = FALSE;
return (result<N-1:0>, saturated);

H1.3.4 shared/translation

shared/translationAttrs/CombineS1S2AttrHints

// CombineS1S2AttrHints()
// ===============
MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;
    if s2desc.attrs == '01' || s1desc.attrs == '01' then
        result.attrs = bits(2) UNKNOWN;   // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
        result.attrs = MemAttr_NC;        // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
        result.attrs = MemAttr_WT;        // Write-through
    else
        result.attrs = MemAttr_WB;        // Write-back
    result.hints = s1desc.hints;
    result.transient = s1desc.transient;
    return result;

shared/translation/attrs/CombineS1S2Desc

// CombineS1S2Desc()
// =============
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;
    result.paddress = s2desc.paddress;
    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        elsif s2desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else               // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device, s2desc.memattrs.device);
        else               // Both Normal
            result.memattrs.type = MemType_Normal;
            result.memattrs.device = DeviceType UNKNOWN;
            result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
            result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
            result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
            result.memattrs.outershareable = (s1desc.memattrs.outershareable || s2desc.memattrs.outershareable);
    result.memattrs = MemAttrDefaults(result.memattrs);
    return result;

shared/translation/attrs/CombineS1S2Device

// CombineS1S2Device()
// ================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_mGnRnE || s1device == DeviceType_mGnRnE then
        result = DeviceType_mGnRnE;
    elsif s2device == DeviceType_mGnRE || s1device == DeviceType_mGnRE then
        result = DeviceType_mGnRE;
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, boolean secondstage)
assert !IsZero(attrfield);
MemAttrHints result;
if !secondstage && PSTATE.EL != EL2 then
    enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
else
    enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
if enable == '0' then
    result.attrs = MemAttr_NC;
    result.hints = MemHint_No;
else
    if attrfield<3:2> == '00' then          // Write-through transient
        result.attrs = MemAttr_WT;
        result.hints = attrfield<1:0>;
        result.transient = TRUE;
    elseif attrfield<3:0> == '0100' then     // Non-cacheable (no allocate)
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
        result.transient = FALSE;
    elseif attrfield<3:2> == '01' then       // Write-back transient
        result.attrs = attrfield<1:0>;
        result.hints = MemAttr_WB;
        result.transient = TRUE;
    else                                    // Write-through/write-back non-transient
        result.attrs = attrfield<3:2>;
        result.hints = attrfield<1:0>;
        result.transient = FALSE;

return result;

---

shared/translation/attrs/MemAttrDefaults

// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)
if memattrs.type == MemType_Device then
    memattrs.inner = MemAttrHints UNKNOWN;
    memattrs.outer = MemAttrHints UNKNOWN;
    memattrs.shareable = TRUE;
    memattrs.outershareable = TRUE;
else
    memattrs.device = DeviceType UNKNOWN;
    if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;

return memattrs;
shared/translation/validation/HasS2Validation

    // HasS2Validation()
    // =================
    // Returns TRUE if stage 2 address validation is present for the current memory
    // access regime

    boolean HasS2Validation()
    {
        return (HaveEL(EL2) && PSTATE.EL IN {EL0, EL1});
    }

shared/translation/validation/S1ValidationRegime

    // S1ValidationRegime()
    // ====================
    // Returns the Exception level controlling the current Stage 1 memory access regime.

    bits(2) S1ValidationRegime()
    {
        if PSTATE_EL != EL0 then
            return PSTATE.EL;
        else
            return EL1;
    }
H1 ARMv8-R AArch32 Pseudocode

H1.3 Shared pseudocode
Part I
Appendixes
This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the ARMv8-R AArch32 architecture. It contains the following sections:

- Reserved values in System registers and memory attribute settings on page I1-354.
I1.1 Reserved values in System registers and memory attribute settings

Unless otherwise stated, all unallocated or reserved values of fields with allocated values in the System registers and memory attribute settings behave in one of the following ways:

• The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.

• The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

• The encoding causes the field to have no functional effect.