Arm® Architecture Reference Manual

for A-profile architecture

ARM DDI 0487H.a (ID020222)

Arm Architecture Reference Manual
for A-profile architecture

Release Information

The following releases of this document have been made.

Release history

Date	Issue	Confidentiality	Change
30 April 2013	A.a-1	Confidential-Beta Draft	Beta draft of first issue, limited circulation
12 June 2013	A.a-2	Confidential-Beta Draft	Second beta draft of first issue, limited circulation
04 September 2013	A.a	Non-Confidential Beta	Beta release
24 December 2013	A.b	Non-Confidential Beta	Second beta release
18 July 2014	A.c	Non-Confidential Beta	Third beta release
09 October 2014	A.d	Non-Confidential Beta	Fourth beta release
17 December 2014	A.e	Non-Confidential Beta	Fifth beta release
25 March 2015	A.f	Non-Confidential Beta	Sixth beta release
10 July 2015	A.g	Non-Confidential Beta	Seventh beta release
30 September 2015	A.h	Non-Confidential Beta	Eighth beta release
28 January 2016	A.i	Non-Confidential Beta	Ninth beta release
03 June 2016	A.j	Non-Confidential EAC	EAC release
30 September 2016	A.k	Non-Confidential Armv8.0 EAC	Updated EAC release
31 March 2017	B.a	Non-Confidential Armv8.1 EAC, v8.2 Beta	Initial release incorporating Armv8.1 and Armv8.2
26 September 2017	B.b	Non-Confidential Armv8.2 EAC	Initial Armv8.2 EAC release, incorporating SPE
20 December 2017	C.a	Non-Confidential Armv8.3 EAC	Initial Armv8.3 EAC release
31 October 2018	D.a	Non-Confidential Armv8.4 EAC	Initial Armv8.4 EAC release
29 April 2019	D.b	Non-Confidential Armv8.4 EAC	Updated Armv8.4 EAC release incorporating accessibility changes
05 July 2019	E.a	Non-Confidential Armv8.5 EAC	Initial Armv8.5 EAC release
20 February 2020	F.a	Non-Confidential Armv8.6 Beta	Initial Armv8.6 Beta release
31 March 2020	F.b	Non-Confidential Armv8.5 EAC, v8.6 Beta	Armv8.5 EAC release, initial Armv8.6 Beta release
17 July 2020	F.c	Non-Confidential Armv8.6 EAC	Initial Armv8.6 EAC release
22 January 2021	G.a	Non-Confidential Armv8.7 EAC	Initial Armv8.7 EAC release
22 July 2021	G.b	Non-Confidential Armv8.7 EAC	Updated Armv8.7 EAC release
4 February 2022	H.a	Non-Confidential Armv9 EAC	Initial Armv9 EAC release

Proprietary Notice

Note

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

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.

The information in this manual is at EAC quality, which means that all features of the specification are described in the manual.

Web Address

`http://www.arm.com`

Limitations of this issue

This issue of the Arm Architecture Reference Manual contains many improvements and corrections. Validation of this document has identified the following issues that Arm will address in future issues:

- Appendix K13 *Arm Pseudocode Definition* requires further review and update. Since this appendix is informative, rather than being part of the architecture specification, this does not affect the quality status of this release.
- For a list of the known issues in this Manual, please refer to the Known Issues document on `https://developer.arm.com/documentation/102105/latest`.
- For a list of the known issues in the System register and instruction XML content, please refer to the Release Notes on `https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools`.

Contents
Arm Architecture Reference Manual for A-profile architecture

Preface

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

About this Manual

This Manual describes the Arm® architecture v8, Armv8, and the Arm® architecture v9, Armv9. The architecture describes the operation of an Armv8-A and an Armv9-A *Processing element (PE)*, and this Manual includes descriptions of:

- The two Execution states, AArch64 and AArch32.
- The instruction sets:
  - In AArch32 state, the A32 and T32 instruction sets, which are compatible with earlier versions of the Arm architecture.
  - In AArch64 state, the A64 instruction set.
- The states that determine how a PE operates, including the current Exception level and Security state, and in AArch32 state the PE mode.
- The Exception model.
- The interprocessing model, that supports transitioning between AArch64 state and AArch32 state.
- The memory model, that defines memory ordering and memory management. This Manual covers the Arm A architecture profile, both Armv8-A and Armv9-A, that defines a *Virtual Memory System Architecture* (VMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, which provide high-performance:
  - Single-precision, half-precision, and double-precision floating-point operations.
  - Conversions between double-precision, single-precision, and half-precision floating-point values.
  - Integer, single-precision floating-point, and half-precision floating-point vector operations in all instruction sets.
  - Double-precision floating-point vector operations in the A64 instruction set.
- The security model, which provides two Security states to support Secure applications.
- The virtualization model.
- The Debug architecture, which provides software access to debug features.

This Manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this Manual is not a tutorial for Arm assembler language, nor does it describe Arm assembler language, except at a very basic level. To make effective use of Arm assembler language, read the documentation supplied with the assembler being used.

This Manual is organized into parts:

Part A Provides an introduction to the Arm architecture, and an overview of the AArch64 and AArch32 Execution states.

Part B Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers' model and the memory model.

Part C Describes the A64 instruction set, which is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as *unprivileged* execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Part D Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers' model and the memory model. This part includes the description of self-hosted debug.

Part E Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers' model and the memory model.

Note

In AArch32 state, execution at EL0 is execution in User mode.

Part F Describes the T32 and A32 instruction sets, which are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the Arm architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Note

User mode is the only mode where software execution is unprivileged.

Part G Describes the system level view of the AArch32 Execution state, which is generally compatible with earlier versions of the Arm architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the instruction interface to those registers. It also describes the system level view of the programmers' model and the memory model.

Part H Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Part I Describes additional features of the architecture that are not closely coupled to a *processing element* (PE), and therefore are accessed through memory-mapped interfaces. Some of these features are OPTIONAL.

Part J Provides pseudocode that describes various features of the Arm architecture.

Part K, Appendixes

Provide additional information. Some appendixes give information that is not part of the Armv8 or Armv9 architectural requirements. The cover page of each appendix indicates its status.

Glossary Defines terms used in this Manual that have a specialized meaning.

Note

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

Using this Manual

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

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-A and Armv9-A architecture profiles, including its relationship to the other Arm PE architectures. It introduces the terminology used to describe the architecture, and gives an overview of the Executions states, AArch64 and AArch32. It contains the following chapters:

Chapter A1 ***Introduction to the Arm Architecture***

Read this for an introduction to the Arm architecture.

Chapter A2 ***Armv8-A Architecture Extensions***

Read this for an introduction to the Armv8 architecture extensions.

Chapter A3 ***Armv9 Architecture Extensions***

Read this for an introduction to the Armv9 architecture extensions.

Part B, The AArch64 Application Level Architecture

Part B describes the AArch64 state application level view of the architecture. It contains the following chapters:

Chapter B1 ***The AArch64 Application Level Programmers' Model***

Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

Chapter B2 ***The AArch64 Application Level Memory Model***

Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The A64 Instruction Set

Part C describes the A64 instruction set, which is used in AArch64 state. It contains the following chapters:

Chapter C1 ***The A64 Instruction Set***

Read this for a description of the A64 instruction set and common instruction operation details.

Chapter C2 ***About the A64 Instruction Descriptions***

Read this to understand the format of the A64 instruction descriptions.

Chapter C3 ***A64 Instruction Set Overview***

Read this for an overview of the A64 instructions.

Chapter C4 ***A64 Instruction Set Encoding***

Read this for a description of the A64 instruction set encoding.

Chapter C5 ***The A64 System Instruction Class***

Read this for a description of the AArch64 System instructions and register descriptions, and the System instruction class encoding space.

Chapter C6 ***A64 Base Instruction Descriptions***

Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Chapter C7 ***A64 Advanced SIMD and Floating-point Instruction Descriptions***

Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Chapter C8 ***SVE Instruction Descriptions***

Read this for information on key aspects of the SVE instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Chapter C9 ***SME Instruction Descriptions***

Read this for information on key aspects of the SME instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Part D, The AArch64 System Level Architecture

Part D describes the AArch64 state system level view of the architecture. It contains the following chapters:

Chapter D1 ***The AArch64 System Level Programmers' Model***

Read this for a description of the AArch64 state system level view of the programmers' model.

Chapter D2 ***AArch64 Self-hosted Debug***

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

Chapter D3 ***AArch64 Self-hosted Trace***

Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

Chapter D4 ***The AArch64 System Level Memory Model***

Read this for a description of the AArch64 state system level view of the general features of the memory system.

Chapter D5 ***The AArch64 Virtual Memory System Architecture***

Read this for a system level view of the AArch64 Virtual Memory System Architecture (VMSA), the memory system architecture of an Armv8 or Armv9 implementation executing in AArch64 state.

Chapter D6 ***The Memory Tagging Extension***

Read this for a description of the Memory Tagging Extension.

Chapter D7 ***The Generic Timer in AArch64 state***

Read this for a description of the AArch64 view of the Arm Generic Timer.

Chapter D8 ***The Performance Monitors Extension***

Read this for a description of the Arm Performance Monitors, an optional non-invasive debug component.

Chapter D9 ***The Activity Monitors Extension***

Read this for a description of the Arm Activity Monitors.

Chapter D10 ***The Statistical Profiling Extension***

Read this for a description of the Statistical Profiling Extension, an optional AArch64 state non-invasive debug component.

Chapter D11 ***Statistical Profiling Extension Sample Record Specification***

Read this for a description of the sample records generated by the Statistical Profiling Extension.

Chapter D12 ***AArch64 System Register Encoding***

Read this for a description of the encoding of the AArch64 System registers, and the other uses of the AArch64 System registers encoding space.

Chapter D13 ***AArch64 System Register Descriptions***

Read this for an introduction to, and description of, each of the AArch64 System registers.

Part E, The AArch32 Application Level Architecture

Part E describes the AArch32 state application level view of the architecture. It contains the following chapters:

Chapter E1 ***The AArch32 Application Level Programmers' Model***

Read this for an application level description of the programmers' model for software executing in AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

Chapter E2 ***The AArch32 Application Level Memory Model***

Read this for an application level description of the memory model for software executing in AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32 state. It includes information about Arm memory types, attributes, and memory access controls.

Part F, The AArch32 Instruction Sets

Part F describes the T32 and A32 instruction sets, which are used in AArch32 state. It contains the following chapters:

Chapter F1 ***About the T32 and A32 Instruction Descriptions***

Read this to understand the format of the T32 and A32 instruction descriptions.

Chapter F2 ***The AArch32 Instruction Sets Overview***

Read this for an overview of the T32 and A32 instruction sets.

Chapter F3 ***T32 Instruction Set Encoding***

Read this for a description of the T32 instruction set encoding. This includes the T32 encoding of the Advanced SIMD and floating-point instructions.

Chapter F4 ***A32 Instruction Set Encoding***

Read this for a description of the A32 instruction set encoding. This includes the A32 encoding of the Advanced SIMD and floating-point instructions.

Chapter F5 ***T32 and A32 Base Instruction Set Instruction Descriptions***

Read this for a description of each of the T32 and A32 base instructions.

Chapter F6 ***T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions***

Read this for a description of each of the T32 and A32 Advanced SIMD and floating-point instructions.

Part G, The AArch32 System Level Architecture

Part G describes the AArch32 state system level view of the architecture. It contains the following chapters:

Chapter G1 ***The AArch32 System Level Programmers' Model***

Read this for a description of the AArch32 state system level view of the programmers' model for execution in an Exception level that is using AArch32.

Chapter G2 ***AArch32 Self-hosted Debug***

Read this for an introduction to, and a description of, self-hosted debug in AArch32 state.

Chapter G3 ***AArch32 Self-hosted Trace***

Read this for an introduction to, and a description of, self-hosted trace in AArch32 state.

Chapter G4 ***The AArch32 System Level Memory Model***

Read this for a system level view of the general features of the memory system.

Chapter G5 ***The AArch32 Virtual Memory System Architecture***

Read this for a description of the AArch32 Virtual Memory System Architecture (VMSA).

Chapter G6 ***The Generic Timer in AArch32 state***

Read this for a description of the AArch32 view of an implementation of the Arm Generic Timer.

Chapter G7 ***AArch32 System Register Encoding***

Read this for a description of the encoding of the AArch32 System registers, including the System instructions that are part of the AArch32 System registers encoding space.

Chapter G8 ***AArch32 System Register Descriptions***

Read this for a description of each of the AArch32 System registers.

Part H, External Debug

Part H describes the architecture for external debug. It contains the following chapters:

Chapter H1 ***About External Debug***

Read this for an introduction to external debug, and a definition of the scope of this part of the Manual.

Chapter H2 ***Debug State***

Read this for a description of Debug state, which the PE might enter as the result of a Halting debug event.

Chapter H3 ***Halting Debug Events***

Read this for a description of the external debug events referred to as Halting debug events.

Chapter H4 ***The Debug Communication Channel and Instruction Transfer Register***

Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

Chapter H5 ***The Embedded Cross-Trigger Interface***

Read this for a description of the embedded cross-trigger interface.

Chapter H6 ***Debug Reset and Powerdown Support***

Read this for a description of reset and powerdown support in the Debug architecture.

Chapter H7 ***The PC Sample-based Profiling Extension***

Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension to an Armv8 or Armv9 implementation.

Chapter H8 ***About the External Debug Registers***

Read this for some additional information about the external debug registers.

Chapter H9 ***External Debug Register Descriptions***

Read this for a description of each external debug register.

Part I, Memory-mapped Components of the Arm architecture

Part I describes the memory-mapped components in the architecture. It contains the following chapters:

Chapter I1 ***Requirements for Memory-mapped Components***

Read this for descriptions of some general requirements for memory-mapped components within a system that complies with the Arm architecture.

Chapter I2 ***System Level Implementation of the Generic Timer***

Read this for a definition of a system level implementation of the Generic Timer.

Chapter I3 ***Recommended External Interface to the Performance Monitors***

Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

Chapter I4 ***Recommended External Interface to the Activity Monitors***

Read this for a description of the recommended memory-mapped interface to the Activity Monitors.

Chapter I5 ***External System Control Register Descriptions***

Read this for a description of each memory-mapped system control register.

Part J, Architectural Pseudocode

Part J contains pseudocode that describes various features of the Arm architecture. It contains the following chapter:

Chapter J1 ***Armv8 Pseudocode***

Read this for the pseudocode definitions that describe various features of the Arm architecture, for operation in AArch64 state and in AArch32 state.

Part K, Appendixes

Note

Some of the descriptions in the following appendixes are not part of the Arm architecture specification. They are included here as supplementary information, for the convenience of developers and users who might require this information.

This Manual contains the following appendixes:

Appendix K1 ***Architectural Constraints on UNPREDICTABLE Behaviors***

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Arm architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

Appendix K2 ***Recommended External Debug Interface***

Read this for a description of the recommended external debug interface.

Appendix K3 ***Recommendations for Reporting Memory Attributes on an Interconnect***

Read this for the Arm recommendations about how the architectural memory attributes are reported on an interconnect.

Appendix K4 ***Additional Information for Implementations of the Generic Timer***

Read this for additional information about implementations of the Arm Generic Timer. This information does not form part of the architectural definition of the Generic Timer.

Appendix K5 ***Legacy Instruction Syntax for AArch32 Instruction Sets***

Read this for information about the pre-UAL syntax of the AArch32 instruction sets, which can still be valid for the A32 instruction set.

Appendix K6 *Address Translation Examples*

Read this for examples of translation table lookups using the translation regimes described in Chapter D5 *The AArch64 Virtual Memory System Architecture* and Chapter G5 *The AArch32 Virtual Memory System Architecture*.

Appendix K7 *Example OS Save and Restore Sequences*

Read this for software examples that perform the OS Save and Restore sequences for an Armv8 or Armv9 debug implementation.

Note

Chapter H6 *Debug Reset and Powerdown Support* describes the OS Save and Restore mechanism.

Appendix K8 *Recommended Upload and Download Processes for External Debug*

Read this for information about implementing and using the Arm architecture.

Appendix K9 *Software Usage Examples*

Read this for software examples that help understanding of some aspects of the Arm architecture.

Appendix K10 *Barrier Litmus Tests*

Read this for examples of the use of barrier instructions provided by the Arm architecture.

Appendix K11 *Random Number Generation*

Read this for information on the generation of random numbers using FEAT_RNG.

Appendix K12 *Legacy Feature Naming Convention*

Read this for an understanding of how the current feature names map to the legacy naming convention.

Appendix K13 *Arm Pseudocode Definition*

Read this for definitions of the Arm pseudocode.

Appendix K14 *Registers Index*

Read this for an alphabetic and functional index of AArch32 and AArch64 registers, and memory-mapped registers.

Glossary

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

Note

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

Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*.
- *Rules-based writing* on page xxvii.
- *Signals* on page xxvii.
- *Numbers* on page xxviii.
- *Pseudocode descriptions* on page xxviii.
- *Assembler syntax descriptions* on page xxviii.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

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

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

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

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the *Glossary*.

Colored text Indicates a link. This can be:

- A URL, for example https://developer.arm.com.
- A cross-reference that includes the page number of the referenced information if it is not on the current page, for example, *Assembler syntax descriptions* on page xxviii.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the Manual that defines the colored term, for example *Simple sequential execution* or SCTLR.

{ and } Braces, { and }, have two distinct uses:

Optional items

In syntax descriptions braces enclose optional items. In the following example they indicate that the `<shift>` parameter is optional:

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

Similarly they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

Sets of items

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE.

Notes Notes are formatted as:

——— **Note** ———

This is a Note.

————————

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note may repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Rules-based writing

Some sections of this Manual use rules-based writing. Rules-based writing consists of a set of individual content items. A content item is classified as one of the following:

- Rule.
- Information.

Rules are normative statements. An implementation that is compliant with this specification must conform to all Rules in this Manual that apply to that implementation.

Rules must not be read in isolation. Where a particular feature is specified by multiple Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read all chapters and sections of this Manual to ensure that an implementation is compliant.

Content items other than Rules are informative statements. These are provided as an aid to understanding this Manual.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this Manual. After content reaches beta status, a given content item has the same identifier across subsequent versions of this Manual.

Content item rendering

In this Manual, a content item is rendered with a token of the following format in the left margin: L_{iiiii}. L is a label that indicates the content class of the content item.

iiiii is the identifier of the content item.

Content item classes

Each of the content item classes has a different function in this Manual.

Rule

A Rule is a statement that describes the behavior of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label R.

Information

An Information statement provides information and guidance as an aid to understanding the Manual.

An Information statement is rendered with the label I.

Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means: • HIGH for active-HIGH signals. • LOW for active-LOW signals.
Lowercase n	At the start or end of a signal name denotes an active-LOW signal.

Numbers

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

Pseudocode descriptions

This Manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in `monospace` font, and is described in Appendix K13 *Arm Pseudocode Definition*.

Assembler syntax descriptions

This Manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in *Structure of the A64 assembler language* on page C1-224, and Appendix K13 *Arm Pseudocode Definition*.

Additional reading

This section lists relevant publications from Arm and third parties.

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

Arm publications

- *ARM® AMBA® 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*, (ARM IHI 0032).
- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for A-profile architecture* (ARM DDI 0587).
- *Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for A-Profile architecture* (ARM DDI 0598).
- *Arm® Architecture Reference Manual Supplement, Armv8, for the Armv8-R AArch32 architecture profile* (ARM DDI 0568).
- *Arm® Architecture Reference Manual Supplement, Armv8, for R-profile AArch64 architecture* (ARM DDI 600).
- *Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile* (ARM DDI 0608).
- *Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* (ARM DDI 0615).
- *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A* (ARM DDI 0616).
- *ARM® Debug Interface Architecture Specification, ADIv6.0* (ARM IHI 0074).
- *ARM® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *ARM® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *ARM® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *ARM® Procedure Call Standard for the ARM 64-bit Architecture* (ARM IHI 0055).

Other publications

The following publications are referred to in this Manual, or provide more information:

- *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001.
- IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
- IEEE Std 754-1985, *IEEE Standard for Floating-point Arithmetic*, March 1985.
- *Secure Hash Standard (SHA)*, Federal Information Processing Standards Publication 180-2, August 2002.
- *The Galois/Counter Mode of Operation*, McGraw, D. and Viega, J., Submission to NIST Modes of Operation Process, January 2004.

- *Memory Consistency Models for Shared Memory-Multiprocessors*, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685.
- *Standard Manufacturer's Identification Code, JEP106*, JEDEC Solid State Technology Association.
- *SM3 Cryptographic Hash Algorithm*, China Internet Network Information Center (CNNIC).
- *SM4 Block Cipher Algorithm*, China Internet Network Information Center (CNNIC).
- *The QARMA Block Cipher Family*, Roberto Avanzi, Qualcomm Product Security Initiative.

  Available from `https://eprint.iacr.org/2016/444`.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Manual

If you have comments on the content of this Manual, send email to `errata@arm.com`. Give:

- The title, *Arm® Architecture Reference Manual, for A-profile architecture*.
- The number, ARM DDI 0487H.a.
- The section name to which your comments refer.
- The page numbers to which your comments refer.
- 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.

Progressive Terminology Commitment

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

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

Part A
Arm Architecture Introduction and Overview

Chapter A1
Introduction to the Arm Architecture

This chapter introduces the Arm architecture. It contains the following sections:

A1.1 About the Arm architecture

The Arm architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a *processing element*, often abbreviated to *PE*. Implementations compliant with the Arm architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to 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 processing element. 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 processing element.

The Arm architecture includes definitions of:

- An associated debug architecture, see:
  - Chapter D2 *AArch64 Self-hosted Debug*.
  - Chapter G2 *AArch32 Self-hosted Debug*.
  - Part H of this Manual, *External Debug* on page Part H-9901.
- Associated trace architectures that define PE Trace Units that implementers can implement with the associated processor hardware. For more information, see:
  - The *Embedded Trace Macrocell Architecture Specification.*
  - Chapter D3 *AArch64 Self-hosted Trace*.
  - Chapter G3 *AArch32 Self-hosted Trace*.

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

- A large uniform register file.
- A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.

This document defines the Armv8-A and Armv9-A architecture *profiles*. See *Architecture profiles* on page A1-38 for more information.

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

An important feature of the Arm architecture is backwards compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases. The Arm architecture supports:

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, that is compatible with previous versions of the Arm architecture.

Note

The AArch32 Execution state is compatible with the Armv7-A architecture profile, and enhances that profile to support some features included in the AArch64 Execution state.

Features that are optional are explicitly defined as such in this Manual.

Note

The presence of an ID register field for a feature does not imply that the feature is optional.

Both Execution states support floating-point instructions:

- AArch32 state provides:
  - SIMD instructions in the base instruction sets that operate on the 32-bit general-purpose registers.
  - Advanced SIMD instructions that operate on registers in the *SIMD and floating-point register* (SIMD&FP register) file.
  - Floating-point instructions that operate on registers in the SIMD&FP register file.
- AArch64 state provides:
  - Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  - Floating-point instructions that operate on registers in the SIMD&FP register file.

    Note

    The A64 instruction set does not include SIMD instructions that operate on the general-purpose registers, therefore, some AArch64 instructions descriptions use SIMD as a synonym for Advanced SIMD.

  - SVE instructions that operate on registers in the SVE register file.

Note

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

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Nine major versions of the architecture have been defined to date, denoted by the version numbers 1 to 9. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

AArch64 Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

Note

The *Base instruction set* comprises the supported instructions other than the floating-point instructions.

See sections *Execution state* on page A1-39 and *The instruction sets* on page A1-40 for more information.

Arm defines three architecture profiles:

A Application profile, described in this Manual:

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

  Note

  An Armv8-A implementation can be called an AArchv8-A implementation and an Armv9-A implementation can be called an AArchv9-A implementation.

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

R Real-time profile:

- 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 designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Implements a variant of the R-profile PMSA.
- Supports a variant of the T32 instruction set.

This Manual describes only Armv8-A and Armv9-A. For information about the R and M architecture profiles, and earlier Arm architecture versions, see:

- The *Arm® Architecture Reference Manual Supplement, Armv8, for the ARMv8-R AArch32 architecture profile.*
- The *Arm® Architecture Reference Manual Supplement, Armv8, for R-profile AArch64 architecture.*
- The *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition.*
- The *Arm®v8-M Architecture Reference Manual.*
- The *ARM®v7-M Architecture Reference Manual.*
- The *ARM®v6-M Architecture Reference Manual.*

A1.3 Arm architectural concepts

This section introduces architectural concepts and the associated terminology.

The following subsections describe key architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:

- *Execution state*.
- *The instruction sets* on page A1-40.
- *System registers* on page A1-40.
- *Arm Debug* on page A1-41.

A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The Exception model.
  - The *Virtual Memory System Architecture* (VMSA).
  - The programmers' model.

The Execution states are:

AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *Program Counter* (PC), *stack pointers* (SPs), and *Exception Link Registers* (ELRs).
- Provides 32 128-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see *The instruction sets* on page A1-40.
- Defines the Armv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an *execution privilege* hierarchy, see *Exception levels* on page D1-4594.
- Provides support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

AArch32 The 32-bit Execution state. This Execution state:

- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and *Link Register* (LR). The LR is used as both an ELR and a procedure link register.
  Some of these registers have multiple *banked* instances for use in different PE *modes*.
- Provides a single ELR, for exception returns from Hyp mode.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, A32 and T32. For more information, see *The instruction sets* on page A1-40.
- Supports the Armv7-A Exception model, based on *PE modes*, and maps this onto the Armv8 Exception model, that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.

- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A32 and T32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the *Application Program Status Register* (APSR) or the *Current Program Status Register* (CPSR).

Later subsections give more information about the different properties of the Execution states.

Transferring control between the AArch64 and AArch32 Execution states is known as *interprocessing*. The PE can move between Execution states only on a change of Exception level, and subject to the rules given in *Interprocessing* on page D1-4647. This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

A1.3.2 The instruction sets

The possible instruction sets depend on the Execution state:

AArch64 AArch64 state supports only a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.

For information on the A64 instruction set, see Chapter C3 *A64 Instruction Set Overview*.

If FEAT_SVE is implemented, the instruction set also supports scalable vector instructions. See *About the SVE instructions* on page C8-2944.

AArch32 AArch32 state supports the following instruction sets:

A32 This is a fixed-length instruction set that uses 32-bit instruction encodings.

T32 This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

In previous documentation, these instruction sets were called the ARM and Thumb instruction sets. Armv8 and Armv9 extend each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.

For information on the A32 and T32 instruction sets, see Chapter F2 *The AArch32 Instruction Sets Overview*.

The instruction sets support SIMD and scalar floating-point instructions. See *Floating-point support* on page A1-56.

A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:

- <register_name>_ELx, where x is 0, 1, 2, or 3.

For information about Exception levels, see *Exception levels* on page D1-4594.

The System registers comprise:

- The following registers that are described in this Manual:
  - General system control registers.
  - Debug registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.
  - Optionally, the Activity Monitors registers.

  - Optionally, the Scalable Vector Extension registers.
- Optionally, one or more of the following groups of registers that are defined in other Arm architecture specifications:
  - Trace System registers, as defined in the *Embedded Trace Macrocell Architecture Specification, ETMv4*.
  - *Generic Interrupt Controller* (GIC) System registers, see *The Arm Generic Interrupt Controller System registers*.
- RAS Extension System registers, as defined in the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*. The RAS Extension is a mandatory extension to the Armv8.2 architecture, and an OPTIONAL extension to the Armv8.0 and the Armv8.1 architectures.

For information about the AArch64 System registers, see Chapter D13 *AArch64 System Register Descriptions*.

For information about the AArch32 System registers, see Chapter G8 *AArch32 System Register Descriptions*.

The Arm Generic Interrupt Controller System registers

From version 3 of the Arm Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this Manual include these registers, see:

- *About the GIC System registers* on page D12-5231, for more information about the AArch64 GIC System registers.
- *About the GIC System registers* on page G7-9002, for more information about the AArch32 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

Note

The programmers' model for earlier versions of the GIC architecture is wholly memory-mapped.

A1.3.4 Arm Debug

Armv8 and later architectures support the following:

Self-hosted debug

In this model, the PE generates *debug exceptions*. Debug exceptions are part of the Armv8 Exception model.

External debug

In this model, *debug events* cause the PE to enter *Debug state*. In Debug state, the PE is controlled by an external debugger.

All Armv8 and later implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:
- In AArch64 state, see Chapter D2 *AArch64 Self-hosted Debug*.
- In AArch32 state, see Chapter G2 *AArch32 Self-hosted Debug*.

For more information about external debug, see Part H *External Debug* on page Part H-9901.

A1.4 Supported data types

The Arm architecture supports the following integer data types:

Byte	8 bits.
Halfword	16 bits.
Word	32 bits.
Doubleword	64 bits.
Quadword	128 bits.

The architecture also supports the following floating-point data types:

- Half-precision, see *Half-precision floating-point formats* on page A1-48 for details.
- Single-precision, see *Single-precision floating-point format* on page A1-50 for details.
- Double-precision, see *Double-precision floating-point format* on page A1-51 for details.
- BFloat16, see *BFloat16 floating-point format* on page A1-52 for details.

It also supports:

- Fixed-point interpretation of words and doublewords. See *Fixed-point format* on page A1-53.
- Vectors, where a register holds multiple elements, each of the same data type. See *Advanced SIMD vector formats* on page A1-43 for details.

The architecture provides two register files:

- A general-purpose register file.
- A SIMD&FP register file.
- If FEAT_SVE is implemented, an SVE scalable vector register file.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- A general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers or as 32-bit registers, using only the bottom 32 bits.
- A SIMD&FP register file contains 128-bit registers:
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see *Instruction Mnemonics* on page C1-225.
- An SVE scalable vector register file contains registers of an IMPLEMENTATION DEFINED width:
  - An SVE scalable vector register has an IMPLEMENTATION DEFINED width that is a multiple of 128 bits, up to a maximum of 2048 bits.
  - All SVE scalable vector registers in an implementation are the same width.
- An SVE predicate register file contains registers of an IMPLEMENTATION DEFINED width:
  - An SVE predicate register has an IMPLEMENTATION DEFINED width that is a multiple of 16 bits, up to a maximum of 256 bits.

For more information on the register files in AArch64 state, see *Registers in AArch64 Execution state* on page B1-135.

In AArch32 state:

- A general-purpose register file contains 32-bit registers:
  - Two 32-bit registers can support a doubleword.
  - Vector formatting is supported, see Figure A1-4 on page A1-46.
- A SIMD&FP register file contains 64-bit registers:
  - AArch32 state does not support quadword integer or floating-point data types.

Note

Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32 state, see *The general-purpose registers, and the PC, in AArch32 state* on page E1-6815.

A1.4.1 Advanced SIMD vector formats

In an implementation that includes the Advanced SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. In AArch32 state, the combination of a register and a data type describes a vector of elements, where the number of elements in the vector is implied by the size of the data type and the size of the register. In AArch64 state, the explicit combination of a register, number of elements, and element size describes a vector of elements.The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant bits of the vector.

For more information on the Advanced SIMD and floating-point registers in AArch32 state, see *The SIMD and floating-point register file* on page E1-6824.

Advanced SIMD vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as V*n*, where *n* is a value from 0 to 31.

The SIMD&FP registers support three data formats for loads, stores, and data-processing operations:

- A single, scalar, element in the least significant bits of the register.
- A 64-bit vector of byte, halfword, word, or doubleword elements.
- A 128-bit vector of byte, halfword, word, or doubleword elements.

The element sizes are defined in Table A1-1 on page A1-43 with the vector format described as:

- For a 128-bit vector: Vn{.2D, .4S, .8H, .16B}.
- For a 64-bit vector: Vn{.1D, .2S, .4H, .8B}.

Table A1-1 SIMD elements in AArch64 state

Mnemonic	Size
`B`	8 bits
`H`	16 bits
`S`	32 bits
`D`	64 bits

Figure A1-1 on page A1-44 shows the SIMD vectors in AArch64 state.

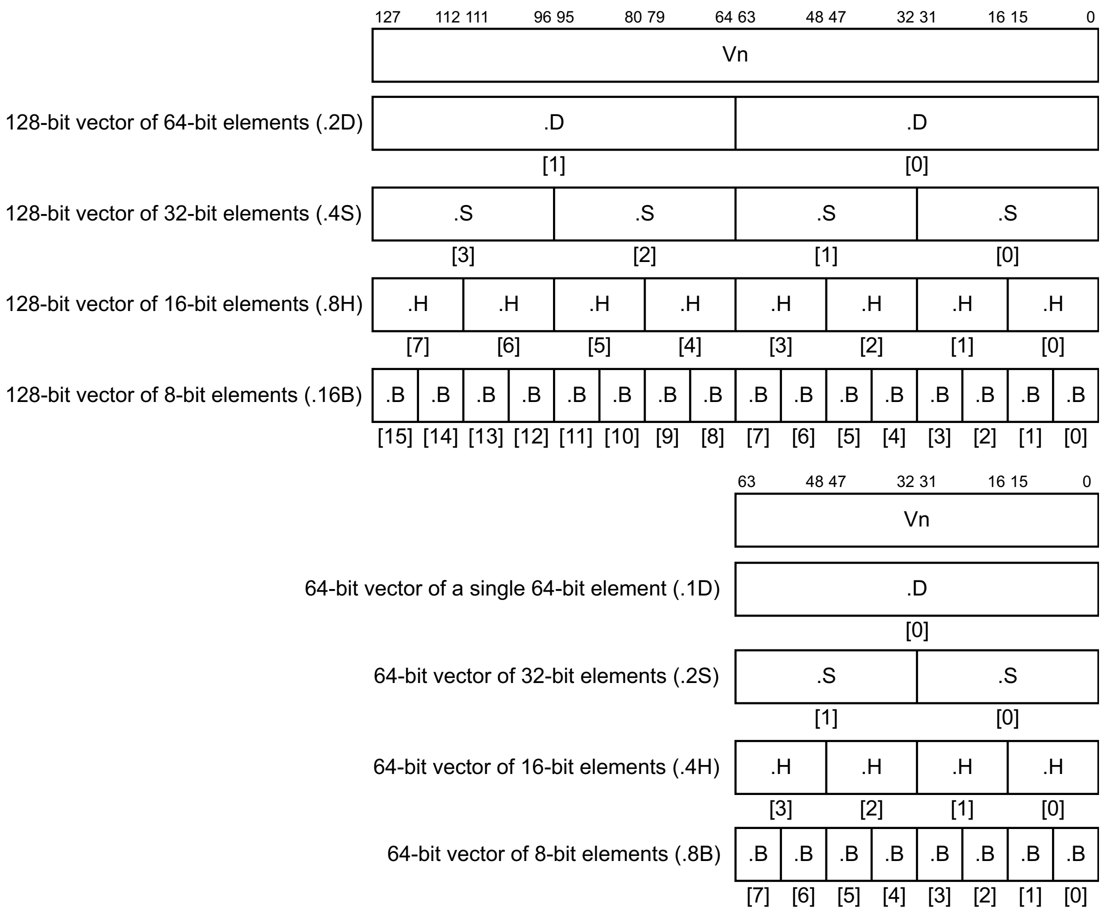

127 112 111 96 95 80 79 64 63 48 47 32 31 16 15 0
Vn
128-bit vector of 64-bit elements (.2D)
.D .D
[1] [0]
128-bit vector of 32-bit elements (.4S)
.S .S .S .S
[3] [2] [1] [0]
128-bit vector of 16-bit elements (.8H)
.H .H .H .H .H .H .H .H
[7] [6] [5] [4] [3] [2] [1] [0]
128-bit vector of 8-bit elements (.16B)
.B .B .B .B .B .B .B .B .B .B .B .B .B .B .B .B
[15] [14] [13] [12] [11] [10] [9] [8] [7] [6] [5] [4] [3] [2] [1] [0]
63 48 47 32 31 16 15 0
Vn
64-bit vector of a single 64-bit element (.1D)
.D
[0]
64-bit vector of 32-bit elements (.2S)
.S .S
[1] [0]
64-bit vector of 16-bit elements (.4H)
.H .H .H .H
[3] [2] [1] [0]
64-bit vector of 8-bit elements (.8B)
.B .B .B .B .B .B .B .B
[7] [6] [5] [4] [3] [2] [1] [0]

Figure A1-1 SIMD vectors in AArch64 state

Advanced SIMD vector formats in AArch32 state

Table A1-2 on page A1-44 shows the available formats. Each instruction description specifies the data types that the instruction supports.

Table A1-2 Advanced SIMD data types in AArch32 state

Data type specifier	Meaning
`.<size>`	Any element of `<size>` bits
`.F<size>`	Floating-point number of `<size>` bits
`.I<size>`	Signed or unsigned integer of `<size>` bits
`.P<size>`	Polynomial over {0, 1} of degree less than `<size>`
`.S<size>`	Signed integer of `<size>` bits
`.U<size>`	Unsigned integer of `<size>` bits

Polynomial arithmetic over {0, 1} on page A1-54 describes the polynomial data type.

The `.F16` data type is the half-precision data type selected by the FPSCR.AHP bit, see *Half-precision floating-point formats* on page A1-48.

The `.F32` data type is the Arm standard single-precision floating-point data type, see *Single-precision floating-point format* on page A1-50.

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-2 shows the hierarchy of the Advanced SIMD data types.

.8	.I8	.S8
		.U8
	.P8	
	-	
.16	.I16	.S16
		.U16
	.P16 †	
	.F16	
.32	.I32	.S32
		.U32
	-	
	.F32	
.64	.I64	.S64
		.U64
	.P64 ‡	
	-	

† Output format only. See `VMULL` instruction description.

‡ Available only if the Cyptographic Extension is implemented.
See `VMULL` instruction description.

Figure A1-2 Advanced SIMD data type hierarchy in AArch32 state

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish between signed and unsigned inputs.

Figure A1-3 on page A1-46 shows the Advanced SIMD vectors in AArch32 state.

——— **Note** ———

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.

———————————

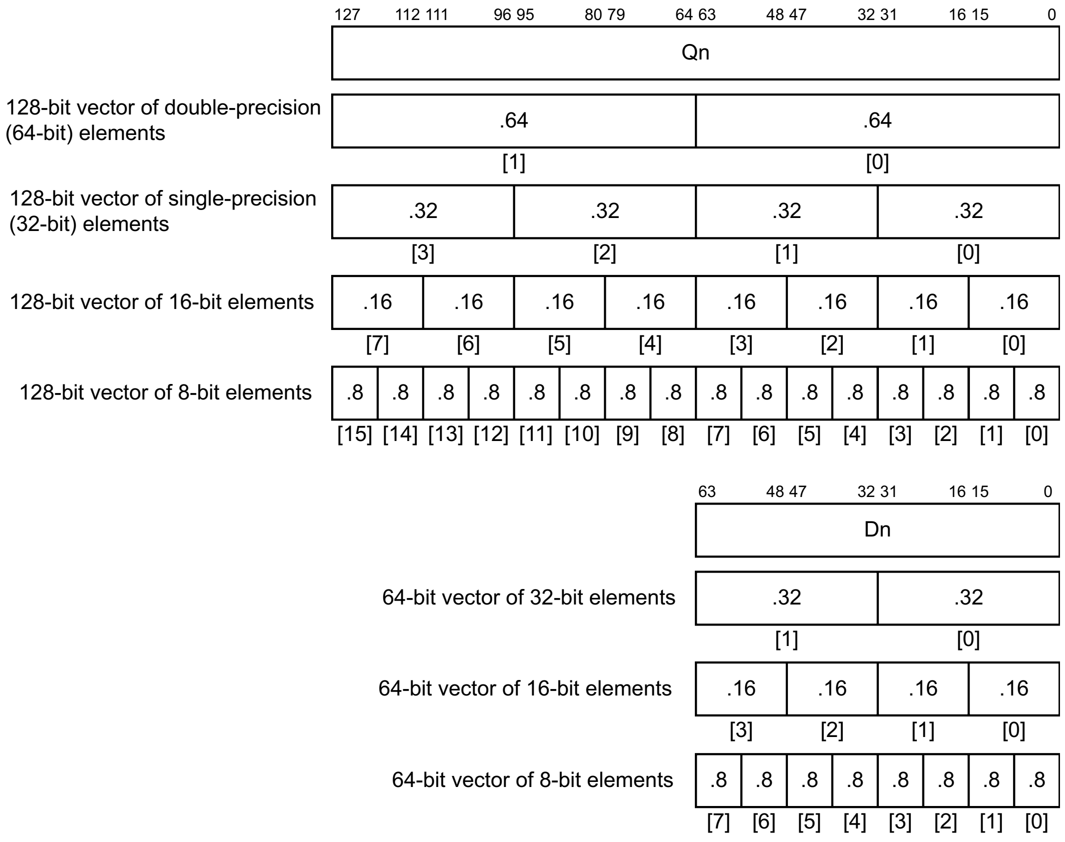


Figure A1-3 Advanced SIMD vectors in AArch32 state

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-4 shows these formats, that means that a general-purpose register can be treated as either 2 halfwords or 4 bytes.

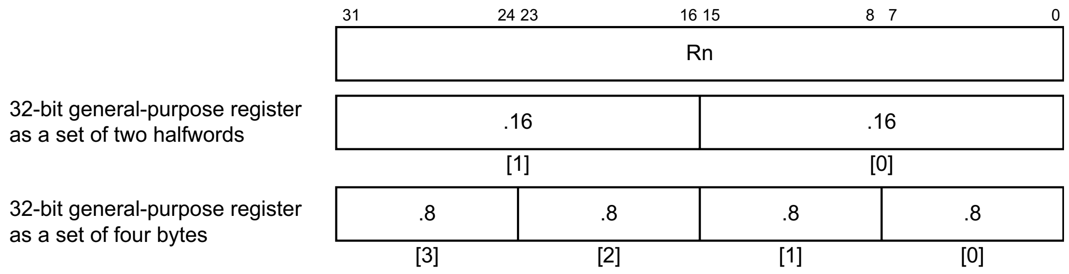

31 24 23 16 15 8 7 0
Rn
32-bit general-purpose register as a set of two halfwords
.16 .16
[1] [0]
32-bit general-purpose register as a set of four bytes
.8 .8 .8 .8
[3] [2] [1] [0]

Figure A1-4 Vector formatting in AArch32 state

A1.4.2 SVE vector format

In an implementation that includes the AArch64 SVE instructions, an SVE register can hold one or more packed or unpacked elements, all of the same size and type. The combination of a register and an element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indexes are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant bits of the vector.

Scalable vector format in AArch64 state

In AArch64 state, the SVE registers can be referred to as Z*n*, where *n* is a value from 0 to 31. For a full description of the SVE scalable vector registers, see *SVE vector registers* on page B1-137.

The element sizes are defined in Table A1-3 with the vector format described as:

- Zn{.Q, .D, .S, .H, .B}.

Table A1-3 SVE elements in AArch64 state

Mnemonic	Size
`B`	8 bits
`H`	16 bits
`S`	32 bits
`D`	64 bits
`Q`	128 bits

SVE configurable vector length

I_{NWYBP} Privileged Exception levels can use the ZCR_ELx.LEN System register fields to constrain the vector length at that Exception level and at less privileged Exception levels.

R_{PVRSF} An implementation allows the vector length to be constrained to any power of two that is less than the maximum implemented vector length.

R_{RYQYY} An implementation is permitted to allow the vector length to be constrained to multiples of 128 that are not a power of two. It is IMPLEMENTATION DEFINED which of the permitted multiples of 128 are supported.

I_{CPZLW} The following table shows the SVE configurable vector lengths:

Maximum	Required	Permitted
128	128	-
256	128, 256	-
384	128, 256	-
512	128, 256	384
640	128, 256, 512	384
768	128, 256, 512	384, 640
896	128, 256, 512	384, 640, 768
1024	128, 256, 512	384, 640, 768, 896
1152	128, 256, 512, 1024	384, 640, 768, 896
1280	128, 256, 512, 1024	384, 640, 768, 896, 1152
1408	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280
1536	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408
1664	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536

Maximum	Required	Permitted
1792	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664
1920	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664, 1792
2048	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664, 1792, 1920

R_{MMCTJ} When the values in ZCR_ELx.LEN configure an unsupported vector length, the implementation is required to select the largest supported vector length that is less than the configured vector length. This does not alter the values in ZCR_ELx.LEN.

R_{PXZTM} If executing at an Exception level that is constrained to use a vector length that is less than the maximum implemented vector length, the bits beyond the constrained length of the vector registers, predicate registers, or FFR are inaccessible.

R_{DMBPN} If floating-point instructions are disabled, trapped, or not available at all Exception levels below the target Exception level, for the current *Security state*, the accessible SVE register state at the target Exception level is preserved.

R_{KXKNK} If any of the following are true and floating-point instructions are not trapped at ELx, then for all purposes other than a direct read, the ZCR_ELx.LEN field has an Effective value of 0, which implies an SVE vector length of 128 bits.

- SVE instructions are disabled at ELx.
- SVE instructions are trapped at ELx.
- SVE instructions are not available because ELx is in AArch32 state.

R_{NLYDK} When taking an exception from an Exception level that is more constrained to a target Exception level that is less constrained, the previously inaccessible bits that become accessible have one of the following:

- A value of zero.
- The value that they had before executing at the more constrained vector length.

The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically.

R_{TQVGX} When the SVE vector length is increased by writing a larger value to ZCR_ELx.LEN, the previously inaccessible bits that become accessible have one of the following:

- A value of zero.
- The value that they had before executing at the more constrained vector size.

The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically.

A1.4.3 Half-precision floating-point formats

The Arm architecture supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.
- Arm *alternative half-precision* format.

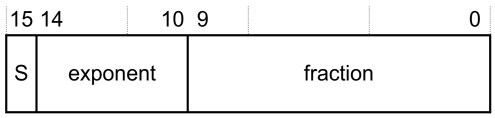
Note

BFloat16 is not a half-precision floating-point format, see *BFloat16 floating-point format* on page A1-52.

Both formats can be used for conversions to and from other floating-point formats. FPCR.AHP controls the format in AArch64 state and FPSCR.AHP controls the format in AArch32 state. FEAT_FP16 adds half-precision data-processing instructions, which always use the IEEE format. These instructions ignore the value of the relevant AHP field, and behave as if it has an *Effective value* of 0. The FEAT_SVE half-precision data-processing instructions ignore the value of FPCR.AHP, and behave as if it has an *Effective value* of 0.

The description of IEEE half-precision includes Arm-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:



The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(\text{exponent-15})} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-14}, or approximately $6.104 \cdot 10^{-5}$.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0	when S==0.
–0	when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-24}, or approximately 5.960×10^{-8}.

Half-precision denormalized numbers are not flushed to zero by default. When FEAT_FP16 is implemented, the FPCR.FZ16 bit controls whether flushing denormalized numbers to zero is enabled for half-precision data-processing instructions. For details, see *Flushing denormalized numbers to zero* on page A1-59.

exponent == `0x1F`

The value depends on which half-precision format is being used:

IEEE half-precision

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity	When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.
-infinity	When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[9]:

bit[9] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

Alternative half-precision

The value is a normalized number and is equal to:

$-1^S \times 2^{16} \times (1.\text{fraction})$

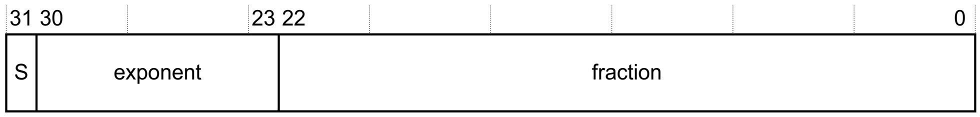
The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

A1.4.4 Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes Arm-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:



The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < `0xFF`

The value is a *normalized number* and is equal to:

$(-1)^S \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a *denormalized number*, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0	When S==0.
–0	When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and –0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see *Flushing denormalized numbers to zero* on page A1-59.

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN*, see *The Default NaN* on page A1-61.

Note

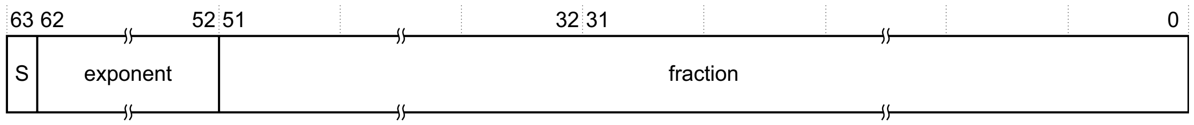
NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.5 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both SIMD and floating-point instructions in AArch64 state, and only by floating-point instructions in AArch32 state.

This description includes implementation-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:



Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < `0x7FF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(\text{exponent}-1023)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0	when S==0.
–0	when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see *Flushing denormalized numbers to zero* on page A1-59.

exponent == `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. As for single-precision, there are two infinities:

+infinity	When S==0.
-infinity	When S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

bit[51] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN*, see *The Default NaN* on page A1-61.

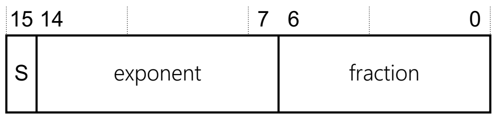
Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.6 BFloat16 floating-point format

BFloat16, or BF16, is a 16-bit floating-point storage format. The BF16 format inherits many of its properties and behaviors from the single-precision format defined by the IEEE 754 standard, as described in *Single-precision floating-point format* on page A1-50.

For the BFloat16 floating-point format, the layout is:



0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(\text{exponent-127})} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately $1.175 \cdot 10^{-38}$.

The maximum positive normalized number is $(2 - 2^{-7}) \times 2^{127}$, or approximately $3.390 \cdot 10^{38}$.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0.

–0 when S==1.

These usually behave identically. However, they yield different results in some circumstances. For example, the sign of the result produced as the result of multiplying by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer bitwise comparison of the two halfwords.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-133}, or approximately 9.184×10^{-41}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see *Flushing denormalized numbers to zero* on page A1-59.

exponent == `0xFF`

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[6]:

bit[6] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[6] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

In the arithmetic instructions that accept BF16 inputs, there is no distinction between quiet and signaling input NaNs, since these instructions cannot signal a floating-point exception, and any type of input NaN generates the same Default NaN result.

BF16 values are 16-bit halfwords that software can convert to single-precision format, by appending 16 zero bits, so that single-precision arithmetic instructions can be used. A single-precision value can be converted to BF16 format if required, either by:

- Truncating, by removing the least significant 16 bits.
- Using the BFloat16 conversion instructions, see *Floating-point single-precision to BFloat16 conversion instruction* on page C3-305.

A1.4.7 Fixed-point format

Fixed-point formats are used only for conversions between floating-point and fixed-point values. They apply to general-purpose registers.

Fixed-point values can be signed or unsigned, and can be 16-bit or 32-bit. Conversion instructions take an argument that specifies the number of fraction bits in the fixed-point number. That is, it specifies the position of the binary point.

A1.4.8 Conversion between floating-point and fixed-point values

The Arm architecture supports the conversion of a scalar floating-point to or from a signed or unsigned fixed-point value in a general-purpose register.

The instruction argument #fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register.

More specifically:

- For a 64-bit register X_d:
  - The integer part is X_d[63:#fbits].
  - The fractional part is X_d[(#fbits-1):0].
- For a 32-bit register W_d or R_d:
  - The integer part is W_d[31:#fbits] or R_d[31:#fbits].
  - The fractional part is W_d[(#fbits-1):0] or R_d[(#fbits-1):0].

These instructions can cause the following floating-point exceptions:

Invalid Operation	When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.
Inexact	When the numeric result differs from the input value.
Input Denormal	When flushing denormalized numbers to zero is enabled and the denormal input is replaced by a zero, see *Flushing denormalized numbers to zero* on page A1-59 and *Input Denormal exceptions* on page A1-66.

Note

An out of range fixed-point result is saturated to the destination size.

For more information, see *Floating-point exceptions and exception traps* on page A1-65.

A1.4.9 Polynomial arithmetic over {0, 1}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over {0, 1}, see *Supported data types* on page A1-42. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit[k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- $0 + 0 = 1 + 1 = 0$.
- $0 + 1 = 1 + 0 = 1$.
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$.
- $1 \times 1 = 1$.

That is:

- Adding two polynomials over {0, 1} is the same as a bitwise exclusive OR.
- Multiplying two polynomials over {0, 1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

A64, A32, and T32 provide instructions for performing polynomial multiplication of 8-bit values.

- For AArch32, see *VMUL (integer and polynomial)* on page F6-8260 and *VMULL (integer and polynomial)* on page F6-8266.

- For AArch64, see *PMUL* on page C7-2511 and *PMULL, PMULL2* on page C7-2513.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See *PMULL, PMULL2* on page C7-2513.

Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function defined in Chapter J1 *Armv8 Pseudocode*.

A1.5 Floating-point support

Note

The architecture includes the following types of floating-point instructions

- Scalar floating-point instructions that operate on the lowest numbered element of the SIMD&FP registers.
- Advanced SIMD floating-point instructions that operate on multiple elements of the SIMD&FP registers.
- If FEAT_SVE is implemented, AArch64 SVE instructions that operate on multiple elements of the scalable vector registers, in which the SIMD&FP registers occupy the least significant 128 bits.

The architecture can support the following levels of support for Advanced SIMD and floating-point instructions:

- Full Advanced SIMD and floating-point support without floating-point exception trapping.
- Full Advanced SIMD and floating-point support with floating-point exception trapping.
- No Advanced SIMD or floating-point support. This option is licensed only for implementations targeting specialized markets.

  Note

  All Armv8-A systems that support standard operating systems with rich application environments provide hardware support for Advanced SIMD and floating-point instructions. All Armv9-A systems that support standard operating systems with rich application environments also provide hardware support for SVE2 instructions. It is a requirement of the ARM Procedure Call Standard for AArch64, see *Procedure Call Standard for the Arm 64-bit Architecture*.

The Arm architecture supports single-precision (32-bit) and double-precision (64-bit) floating-point data types and arithmetic as defined by the IEEE 754 floating-point standard. It also supports the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types and double-precision and half-precision data types. When FEAT_FP16 is implemented, it also supports the half-precision floating-point data type for data-processing operations.

The SIMD instructions provide packed *Single Instruction Multiple Data (SIMD)* and single-element scalar operations, and support:

- Single-precision and double-precision arithmetic in AArch64 state.
- Single-precision arithmetic only in AArch32 state.
- When FEAT_FP16 is implemented, half-precision arithmetic is supported in AArch64 and AArch32 states.

Floating-point support in AArch64 state SIMD is IEEE 754-2008 compliant with:

- Configurable rounding modes.
- Configurable Default NaN behavior.
- Configurable flushing to zero of denormalized numbers.

Floating-point computation using AArch32 Advanced SIMD instructions remains unchanged from Armv7. A32 and T32 Advanced SIMD floating-point always uses Arm standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see *Flushing denormalized numbers to zero* on page A1-59.
- Only default NaNs are supported, see *The Default NaN* on page A1-61.
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

If floating-point exception trapping is supported, floating-point exceptions, such as Overflow or Divide by Zero, can be handled without trapping. This applies to both SIMD and floating-point operations. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. For more information about floating-point exceptions, see *Floating-point exceptions and exception traps* on page A1-65.

In AArch64 state, the following registers control floating-point operation and return floating-point status information:

- The Floating-Point Control Register, FPCR, controls:
  - The half-precision format where applicable, FPCR.AHP bit.
  - Default NaN behavior, FPCR.DN bit.
  - Flushing of denormalized numbers to zero, FPCR.{FZ, FZ16} bits. If FEAT_FP16 is not implemented, FPCR.FZ16 is RES0.
  - Rounding mode support, FPCR.Rmode field.
  - Len and Stride fields associated with execution in AArch32 state, and only supported for a context save and restore from AArch64 state. These fields are obsolete in Armv8 and can be implemented as RAZ/WI. If they are implemented as RW and are programmed to a nonzero value, they make some AArch32 floating-point instructions UNDEFINED.
  - Floating-point exception trap controls, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits, see *Floating-point exceptions and exception traps* on page A1-65.
- The Floating-Point Status Register, FPSR, provides:
  - Cumulative floating-point exceptions flags, FPSR.{IDC, IXC, UFC, OFC, DZC, IOC and QC}.
  - The AArch32 floating-point comparison flags {N,Z,C,V}. These bits are RES0 if AArch32 floating-point is not implemented.

    Note

    In AArch64 state, the process state flags, PSTATE.{N,Z,C,V} are used for all data-processing compares and any associated conditional execution.

    If FEAT_FlagM2 is implemented, the instructions `AXFLAG` and `XAFLAG` convert between the Arm condition flag format and an alternative format shown in *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-1142.

AArch32 state provides a single Floating-Point Status and Control Register, FPSCR, combining the FPCR and FPSR fields.

For system level information about the SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-8680.

A1.5.1 Instruction support

The floating-point instructions support:

- Load and store for single elements and vectors of multiple elements.

  Note

  Single elements are also referred to as scalar elements.

- Data processing on single and multiple elements for both integer and floating-point data types.
- When FEAT_FCMA is implemented, complex number arithmetic.
- Floating-point conversion between different levels of precision.
- Conversion between floating-point, fixed-point integer, and integer data types.
- Floating-point rounding.

For more information on floating-point instructions in AArch64 state, see Chapter C3 *A64 Instruction Set Overview*.

For more information on floating-point instructions in AArch32 state, see Chapter F2 *The AArch32 Instruction Sets Overview*.

A1.5.2 Floating-point standards, and terminology

The Arm architecture includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this Manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- Arm floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-4 on page A1-58 shows how the terminology in this Manual differs from that used in IEEE 754-2008.

Table A1-4 Floating-point terminology

This manual	IEEE 754-2008
Normalized [a]	Normal
Denormal, or denormalized	Subnormal
Round towards Minus Infinity (RM)	roundTowardsNegative
Round towards Plus Infinity (RP)	roundTowardsPositive
Round towards Zero (RZ)	roundTowardZero
Round to Nearest (RN)	roundTiesToEven
Round to Nearest with Ties to Away	roundTiesToAway
Rounding mode	Rounding-direction attribute

a. *Normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this Manual.

A1.5.3 Arm standard floating-point input and output values

The Arm architecture provides full IEEE 754 floating-point arithmetic support. In AArch32 state, floating-point operations performed using Advanced SIMD instructions are limited to *Arm standard floating-point operation*, regardless of the selected rounding mode in the FPSCR. Unlike AArch32, AArch64 SIMD floating point arithmetic is performed using the rounding mode selected by the FPCR.

Arm standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see *Flushing denormalized numbers to zero* on page A1-59.
- NaNs.
- Infinities.

Arm standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

Arm standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results that are less than the minimum normalized number are flushed to zero, see *Flushing denormalized numbers to zero* on page A1-59.
- NaNs produced in floating-point operations are always the default NaN, see *The Default NaN* on page A1-61.
- Infinities.

A1.5.4 Flushing denormalized numbers to zero

For this section if FEAT_AFP is not implemented, the behavior is the same as if FPCR.AH == 0, FPCR.FZ == 0 and FPCR.NEP == 0.

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and Intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization.

If a number `value` satisfies the condition `0 < Abs(value) < MinNorm`, it is treated as a denormalized number.

`MinNorm` is defined as follows:

- For half-precision numbers, `MinNorm` is 2^{-14}.
- For single-precision and BFloat16 numbers, `MinNorm` is 2^{-126}.
- For double-precision numbers, `MinNorm` is 2^{-1022}.

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.
- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has an input denormalized number that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For floating-point instructions, if the instruction processes half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If FPCR.FZ16 == 0, denormalized half-precision inputs are not flushed to zero.
- If FPCR.FZ16 == 1, for half-precision data-processing instructions, flushing of input denormalized numbers to zero occurs as follows:
  - If an instruction does not convert a half-precision input to a higher precision output, all input denormalized numbers are flushed to zero.
  - If an instruction converts a half-precision input to a higher precision output, input denormalized numbers are not flushed to zero.

If FPCR.FIZ == 1, or both FPCR.AH == 0 and FPCR.FZ == 1, for Advanced SIMD and floating-point instructions other than FABS and FNEG, all single-precision, double-precision and BF16 input operands that are denormalized numbers are flushed to zero. Half-precision input operands are not flushed to zero.

If FPCR.FZ == 0, for Advanced SIMD, floating-point and BF16 instructions, for single-precision, double-precision and BF16 inputs, the FPCR.FZ setting does not cause denormalized inputs to be flushed to zero, although other factors might cause denormalized numbers to be flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FIZ, all of the following instructions flush all input denormal numbers to zero:

- BFloat instructions: BFCVT, BFCVTN, BFCVTN2, BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), and BFCVTNT.

- Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

Flushing to zero of denormalized numbers as Intermediate results of some BF16 instructions

BF16 arithmetic instructions BFDOT (by element), BFDOT (vector), BFMMLA in AArch64, and VDOT (by element), VDOT (vector), VMMLA in AArch32 when working with BF16 inputs, convert BF16 input values to IEEE single-precision format, and calculate N-way dot-products, accumulating the products in single-precision accumulators.

If a BF16 arithmetic instruction processes an Intermediate result that is a single-precision denormalized number, the Intermediate result is unconditionally flushed to zero.

Flushing denormalized outputs to zero

If a denormalized output is flushed to zero, the output is returned as zero with the same sign bit as the denormalized output value.

If FPCR.AH == 0, for half-precision, single-precision and double-precision numbers, the test for a denormalized number for the purpose of flushing the output to zero occurs before rounding.

If FPCR.AH == 1, and if output flushing is caused by FPCR.FZ == 1 or FPCR.FZ16 == 1, for half-precision, single-precision and double-precision numbers, the test for a denormalized number for the purpose of flushing the output to zero occurs after rounding using an unbounded exponent.

If FPCR.AH == 1, and if FPCR.FZ == 0, Advanced SIMD, floating-point and BF16 instructions, for single-precision, double-precision and BF16 outputs, the FPCR.FZ setting does not cause denormalized outputs to be flushed to zero, although other factors might cause denormalized outputs to be flushed to zero.

BFDOT (by element), BFDOT (vector), and BFMMLA instructions unconditionally flush denormalized output numbers to zero.

If FPCR.AH == 0, for Advanced SIMD, floating-point, and BF16 instructions, for single-precision, double-precision and BF16 outputs, flushing denormalized numbers to zero can be controlled as follows:

- If FPCR.FZ == 0, the FPCR.FZ setting does not cause denormalized output numbers to be flushed to zero, although other factors might cause denormalized output numbers to be flushed to zero.
- If FPCR.FZ == 1, for all Advanced SIMD, floating-point and BF16 instructions other than `FABS` and `FNEG`, all single-precision, double-precision, and BF16 outputs that are denormalized numbers are flushed to zero.

If FPCR.FZ16 == 0 denormalized half-precision output numbers are not flushed to zero.

If FPCR.FZ16 == 1, for floating-point instructions other than `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, if the instruction processes half-precision numbers, flushing denormalized output numbers to zero can be controlled as follows:

- Instructions that convert between half-precision and single-precision numbers do not flush denormalized half-precision output numbers to zero.
- Instructions that convert between half-precision and double-precision numbers do not flush denormalized half-precision output numbers to zero.
- All other half-precision data-processing instructions flush all denormalized half-precision output numbers to zero.

If FPCR.AH == 1 and FPCR.FZ == 1, for Advanced SIMD, floating-point and BF16 instructions, all of the following apply:

- For all floating-point operations other than `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, all single-precision and double-precision denormalized output operands are flushed to zero.
- For `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, denormalized output operands are not flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FZ bit, for both Advanced SIMD and SVE, all of the following instructions flush all output denormal numbers to zero:

- BFloat instructions: BFCVT, BFCVTN, BFCVTN2, BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), and BFCVTNT.
- Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

A1.5.5 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the mantissa. The Arm architecture additionally defines a Default NaN which does not follow this format.

The IEEE 754 standard specifies that the sign bit of a NaN has no significance.

For a quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a speculative result of a floating-point calculation not involving NaN inputs.

The Default NaN

The Default NaN is encoded as described inTable A1-5 on page A1-61.

Table A1-5 Default NaN encoding

	Half-precision, IEEE Format	Single-precision	Double-precision	BFloat16
Sign bit If FPCR.AH == 0	0	0	0	0
Sign bit If FPCR.AH == 1	1	1	1	1
Exponent	`0x1F`	`0xFF`	`0x7FF`	`0xFF`
Fraction	Bit[9] == 1, bits[8:0] == 0	Bit[22] == 1, bits[21:0] == 0	Bit[51] == 1, bits[50:0] == 0	Bit[6] == 1, bits[5:0] == 0

IF FPCR.DN == 1, for floating-point instructions other than FABS, FMAX* FMIN* and FNEG, if any input to a floating-point operation performed by the instruction is a NaN, the output of the floating-point operation is the Default NaN.

For `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, Default NaN behavior is explained in the instruction description.

If FPCR.DN == 0, for floating-point processing the Default NaN is not used for NaN propagation.

If a floating-point instruction performs a floating-point operation, and that instruction generates an untrapped Invalid Operation floating-point exception for a reason other than one of the inputs being a signaling NaN, the output is the Default NaN.

NaN handling

The IEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived quiet NaN output might have both a different sign and a different number of fraction bits from its source. See instruction descriptions for details.

NaN propagation

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the instruction and the number of operands.

If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN.
- If the input NaN is a signaling NaN, the output NaN is derived as follows:
  - If the handling of a signaling NaN by the instruction generates an Invalid Operation exception, the output NaN is the quieted version of the input NaN.
  - If the handling of a signaling NaN by the instruction does not generate an Invalid Operation exception, the output NaN is the same as the input NaN. This case applies for FABS, FNEG, and FTSSEL instructions.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, the term "first operand" and "second operand" relate to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If FPCR.DN == 0, for Advanced SIMD, floating-point, or BF16 instructions that perform a floating-point operation, other than `FABS`, `FNEG`, `FMAX*`, and `FMIN*`, NaN outputs that derive from NaN inputs are derived as follows:

- If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:
  - FPCR.AH == 0.
  - At least one operand is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, an instruction outputs a quiet NaN derived from the first NaN operand:
  - FPCR.AH == 0.
  - At least one operand is a NaN, but none of the operands is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, the output is a quiet NaN derived from the NaN operand:
  - FPCR.AH == 1.
  - The operation has two floating-point inputs.
  - The operation has only one NaN operand.
- If all of the following apply, the output is a NaN derived from the `<Vn>`, `<Hn>`, `<Sn>`, or `<Dn>` register:
  - FPCR.AH == 1.
  - The operation has two floating-point inputs.
  - The operation has two NaN operands.

- If all of the following apply, the output is a NaN derived from the NaN held in the `<Vn>`, `<Hn>`, `<Sn>`, or `<Dn>` register:
  - FPCR.AH == 1
  - The instruction is one of: BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), FCMLA, FMADD, FMLA (by element), FMLA (vector), FMLAL, FMLAL2 (by element), FMLAL, FMLAL2 (vector), FMLS (by element), FMLS (vector), FMLSL, FMLSL2 (by element), FMLSL, FMLSL2 (vector), FMSUB, FNMADD, and FNMSUB.
  - One of the following applies:
    - The operation has three NaN operands.
    - The operation has two NaN operands and the `<Vn>`, `<Hn>`, `<Sn>` or `<Dn>` register holds a NaN.
- If all of the following apply, the output is a NaN derived from the NaN held in the `<Vm>`, `<Hm>`, `<Sm>`, or `<Dm>` register:
  - FPCR.AH == 1
  - The instruction is one of: BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), FCMLA, FMADD, FMLA (by element), FMLA (vector), FMLAL, FMLAL2 (by element), FMLAL, FMLAL2 (vector), FMLS (by element), FMLS (vector), FMLSL, FMLSL2 (by element), FMLSL, FMLSL2 (vector), FMSUB, FNMADD, and FNMSUB.
  - The operation has two NaN operands and the `<Vn>`, `<Hn>`, `<Sn>` or `<Dn>` register does not hold a NaN.

If FPCR.AH == 0, and an output NaN is derived from an input NaN, the pseudocode functions `FPAbs()`, `FPNeg()`, `FPTrigMAdd()`, and `FPTrigSSel()` can change the sign of the NaN,

If FPCR.AH == 1, and an output NaN is derived from an input NaN, for all cases, the sign bit of the NaN is unchanged.

For `FMAX*` and `FMIN*`, the NaN handling is described in the instruction.

A1.5.6 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPCR.RMode or by the instruction.

If FPCR.AH == 1, for any value of FPCR.RMode, the following instructions use Round to Nearest on outputs:

- BF16 instructions BFCVT, BFCVTN, BFCVTN2, BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), and the SVE instruction BFCVTNT.
- Single-precision and double-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.
- Half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

The rounding mode control field FPCR.RMode can select the following rounding modes:

- Round to Nearest (RN) mode.
- Round towards Plus Infinity (RP) mode.
- Round towards Minus Infinity (RM) mode.
- Round towards Zero (RZ) mode.

The following two additional rounding modes are not selected by FPCR.RMode, but are used by some instructions:

- Round to Odd mode.
- Round to Nearest with ties to away mode.

Round to Nearest mode

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round towards Plus Infinity mode

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

Round towards Minus Infinity mode

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

Round towards Zero mode

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounding mode is used by the FCVTAS (scalar), FCVTAS (vector), FCVTAU (scalar), FCVTAU (vector), FRINTA (scalar), and FRINTA (vector) instructions.

Round to Nearest with Ties to Away rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity, the sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the larger number.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round to Odd mode

Round to Odd mode is not defined by IEEE 754, and differs between the FCVTXN, FCVTXN2 instructions, and the BFDOT (by element), BFDOT (vector), and BFMMLA instructions.

The FCVTXN, FCVTXN2 instructions use Round to Odd rounding mode. If the result of the rounding is inexact, the least significant bit of the mantissa is forced to 1.

Round to Odd rounding mode can avoid double rounding errors when a floating-point value is converted to a lower precision destination format through an intermediate precision format.

Example A1-1 Converting 64-bit floating-point format to 16-bit floating-point format

---

A 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value using the following steps:

1. Use an `FCVTXN` instruction to produce a 32-bit value.
2. Use another instruction with the required rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

For BFDOT (by element), BFDOT (vector), and BFMMLA instructions, if the intermediate format has at least two more bits of precision than the result format, Round to Odd mode is used and operates as follows:

- If the rounded value is inexact, the least significant bit of the fraction is set to 1.
- If the value is too large to represent in the single-precision format, the rounded value is a single-precision Infinity, the sign of the rounded value is the same as the sign of the value before rounding.

A1.5.7 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*.

Predicated SVE floating-point instructions only generate floating-point exceptions in response to floating-point operations performed on *Active elements*.

Note

In AArch64 state, an Advanced SIMD or SVE instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an *operation*, rather than on an *instruction*.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

Synchronous exception generation by floating-point exceptions is enabled using the FPCR as follows:

- For each floating-point exception that has synchronous exception generation supported, the relevant control bits chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are used to enable synchronous exception generation.
- For each floating-point exception that does not have synchronous exception generation supported, the relevant bits chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are RAZ/WI.

Input Denormal exceptions

The cumulative floating-point exception bit FPSR.IDC, and the trap enable bit FPCR.IDE both relate to Input Denormal exceptions.

If an input denormalized number is flushed to zero, the occurrence of the Input Denormal exception is determined using the value before flushing.

If an input denormalized number is flushed to zero, and FPCR.AH is 0, the occurrence of all floating-point exceptions, except Input Denormal, is determined treating the input value that is flushed to zero as zero.

If an input denormalized number is flushed to zero, and FPCR.AH is 1, the occurrence of all floating-point exceptions is determined treating the input value that is flushed to zero as zero.

If FPCR.AH is 0, when a single-precision or double-precision floating-point input is flushed to zero, an Input Denormal exception is generated.

If FPCR.AH is 1, and FPCR.FIZ is 0, if and only if none of the following apply, any operation that unpacks a denormalized floating-point input, other than unpacking a BFloat or half-precision value, generates an Input Denormal exception:

- One of the other operands of the instruction is a NaN.
- The operation generates an Invalid Operation floating-point exception.
- The operation generates a Divide-by-Zero floating-point exception.
- The instruction that generated the operation was one of: BFCVTN, BFCVTN2, BFCVT, and BFCVTNT.
- The denormalized floating-point input is flushed to zero.

When a half-precision floating-point value is flushed to zero, an Input Denormal exception is not generated.

If FPCR.AH is 1, or FPCR.FZ is 0, when FPCR.FIZ causes flushing of a denormalized number, an Input Denormal Exception is not generated.

Inexact exceptions

The cumulative floating-point exception bit FPSR.IXC and the trap enable bit FPCR.IXE both relate to Inexact exceptions.

If a denormalized output is flushed to zero, all of the following apply:

- If FPCR.AH is 1, an Inexact exception is generated.
- If FPCR.AH is 0, an Inexact exception is not generated.

If a result is not flushed to zero, and the result does not equal the result computed with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

Underflow exceptions

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPCR.UFE both relate to Underflow exceptions.

If FPCR.AH is 1, for all floating points other than `BFMul`() and `BFAdd`() which are used by `BFDOT` and `BFMLLA`, for the purpose of underflow floating-point exception generation, a denormalized number is detected after rounding with an unbounded exponent.

If FPCR.AH is 0, for the purpose of underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then:

- If FPCR.UFE is 0, and the result is inexact, then the underflow floating-point exception is generated.
- If FPCR.UFE is 1, then the underflow floating-point exception is generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is not generated.

Overflow exceptions

The cumulative floating-point exception bit FPSR.OFC, and the trap enable bit FPCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the rounding mode and the sign of the result before rounding as follows:

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision's largest finite number with the sign of the result before rounding.

Divide by Zero exceptions

The cumulative floating-point exception bit FPSR.DZC, and the trap enable bit FPCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite non-zero number by zero, a Divide by Zero exception is generated.

If a floating-point operation divides a finite non-zero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

Invalid Operation exceptions

The cumulative floating-point exception bit FPSR.IOC, and the trap enable bit FPCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN.
- Magnitude subtraction of infinities.
- Multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Square root of an operand that is less than zero.

If the input is one of: a quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation exception.

For the signaling compare instructions FCMPE and FCCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

If FPCR.AH is 1, for FMAX (vector), FMAX (scalar), FMAXP (scalar), FMAXP (vector), FMAXV, FMIN (vector), FMIN (scalar), FMINP (scalar), FMINP (vector), and FMINV, if either input is any type of NaN, then an Invalid Operation floating-point exception is generated.

Operations that do not generate floating point exceptions

BFDOT (by element), BFDOT (vector), and BFMMLA do not generate floating-point exceptions.

If FPCR.AH is 1, all of the following instructions do not generate any floating-point exceptions regardless of their input values:

- BF16 instructions BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), BFCVT, BFCVTN, BFCVTN2, and BFCVTNT.
- Single-precision, double-precision and half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.
- Floating-point to integer and floating-point rounding instructions: FCVTMS (scalar), FCVTMS (vector), FCVTMU (scalar), FCVTMU (vector), FCVTNS (scalar), FCVTNS (vector), `FCVTNU (scalar)`, FCVTNU (vector), FCVTPS (scalar), FCVTPS (vector), FCVTPU (scalar), FCVTPU (vector), FCVTZS (scalar, fixed-point), FCVTZS (scalar, integer), FCVTZS (vector, fixed-point), FCVTZS (vector, integer), FCVTZU (scalar, fixed-point), FCVTZU (scalar, integer), FCVTZU (vector, fixed-point), FCVTAS (scalar), FCVTAS (vector), FCVTAU (scalar), FCVTAU (vector), FCVTZS (scalar, fixed-point), FCVTZS (scalar, integer), FCVTZS (vector, fixed-point), FCVTZS (vector, integer), FRINTA (scalar), FRINTA (vector), FRINTZ (scalar), FRINTZ (vector), FRINTM (scalar), FRINTM (vector), FRINTP (scalar), FRINTP (vector), FRINTN (scalar), FRINTN (vector), FRINTX (scalar), FRINTX (vector), FRINTI (scalar), FRINTI (vector), FRINT32X (scalar), FRINT32X (vector), FRINT32Z (scalar), FRINT32Z (vector), FRINT64X (scalar), FRINT64X (vector), FRINT64Z (scalar), and FRINT64Z (vector).

`FPAbs()` and `FPNeg()` are not classified as floating-point operations and all of the following apply to them:

- They cannot generate floating-point exceptions.
- The floating-point behavior described in the *Flushing denormalized numbers to zero* on page A1-59 does not apply to them.
- The floating-point behavior described in the section*NaN handling and the Default NaN* on page A1-61 does not apply to them.

Handling floating-point exceptions

If generating synchronous exceptions is enabled for one or more floating-point exceptions, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception and that is not at a lower Exception level than where the exception was generated.

If an implementation includes synchronous exception generation for floating-point exceptions in AArch64 state, all of the following apply:

- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception, except that an implementation is permitted to not restore the cumulative floating-point exception bits in the event of such an exception.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported.

The `AArch64.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.

Combinations of floating-point exceptions

More than one floating-point exception can occur on the same operation. The only combinations of floating-point exceptions that can occur are:

- Overflow with Inexact.
- Underflow with Inexact.
- If FPCR.AH is 0, Input Denormal with any other floating-point exceptions.
- If FPCR.AH is 1, Input Denormal with Inexact, Underflow, or Overflow.

If two floating-point exceptions occur on the same operation, the Input Denormal exception is treated as highest priority and the Inexact exception is treated as lowest priority.

Some floating-point instructions specify more than one floating-point operation, this is indicated by the pseudocode descriptions of the instruction. In these cases, it is possible for one instruction to generate multiple exceptions. Multiple exceptions from one instruction are prioritized as follows:

- If an exception generating operation outputs a result that is used by a second exception generating operation, the exception of the operation that outputs the result is treated as higher priority than the exception of the second operation that uses the result.
- If exception generating operations do not use the outputs of other exception generating operations, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority. The exception prioritized might differ between different instances of the same two floating-point exceptions being generated on the same operation during execution of the instruction.
- A trapped underflow floating-point exception has priority over a trapped inexact floating-point exception.

If none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative bit in the FPSR to be set to 1.

When a floating-point exception is trapped, all of the following apply:

- When the trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value of the FPSR immediately before the instruction that generated the trapped floating-point exception.

  When the trapped floating-point exception is taken, if the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which untrapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative floating-point exception bits having the value 1.
- In the ESR_ELx to which the trapped exception is taken all of the following apply:
  - The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
  - If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.
  - If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

The architectural requirements for floating-point exception prioritization apply only to multiple floating-point exceptions generated on the same element of an Advanced SIMD or SVE operation. For trapped floating-point exceptions from Advanced SIMD or SVE instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.

A1.6 The Arm memory model

The Arm memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating *virtual addresses* (VAs) provided by executing instructions to *physical addresses* (PAs).
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.
- Barriers that control and prevent speculative access to memory.

VA support depends on the Execution state, as follows:

AArch64 state

Supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

AArch32 state

Supports 32-bit virtual addressing, with the Translation Control Register determining the supported VA range. For execution at EL1 and EL0, system software can split the VA range into two subranges, each with its own translation controls.

The supported PA space is IMPLEMENTATION DEFINED, and can be discovered by system software.

Regardless of the Execution state, the *Virtual Memory System Architecture* (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported PA space.

For more information, see:

For execution in AArch64 state

- Chapter B2 *The AArch64 Application Level Memory Model*.
- Chapter D4 *The AArch64 System Level Memory Model*.
- Chapter D5 *The AArch64 Virtual Memory System Architecture*.

For execution in AArch32 state

- Chapter E2 *The AArch32 Application Level Memory Model*.
- Chapter G4 *The AArch32 System Level Memory Model*.
- Chapter G5 *The AArch32 Virtual Memory System Architecture*.

Chapter A2
Armv8-A Architecture Extensions

This chapter introduces the Arm architecture versions and extensions. It contains the following sections:

A2.1 Armv8.0 architecture extensions

The original Armv8-A architecture is called Armv8.0. The following sections of this manual describe or summarize permitted extensions to Armv8.0:

- *The Armv8 Cryptographic Extension* on page A2-80.
- *The Reliability, Availability, and Serviceability Extension* on page A2-120.
- *Event monitors* on page D1-4646.
- *The IVIPT Extension* on page D5-4933.
- Chapter H7 *The PC Sample-based Profiling Extension*.

Note

The naming convention of features in the Arm architecture has been redefined. For more information on how these names map to the legacy convention, see Appendix K12 *Legacy Feature Naming Convention*.

In addition to describing Armv8.0, this manual describes the following architectural extensions:

Features added to Armv8.0 in later releases

Architectural features and architectural requirements have been added to the original Armv8-A architecture. For more information, see:

- *Additional functionality added to Armv8.0 in later releases* on page A2-76.
- *Architectural requirements within Armv8.0 architecture* on page A2-79.

For more information, see *Architectural features within Armv8.0 architecture* on page A2-76.

The Armv8.1 architectural extension

The Armv8.1 architecture extension adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.1 compliant if all of the following apply:

- It includes all of the Armv8.1 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.1 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.1* on page A2-82 for all of the Armv8.1 architectural features.
- It includes all of the Armv8.1 architectural requirements. *Additional requirements of Armv8.1* on page A2-84 lists these requirements.

For more information, see *The Armv8.1 architecture extension* on page A2-82.

The Armv8.2 architectural extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.2 compliant if all of the following apply:

- It is Armv8.1 compliant.
- It includes all of the Armv8.2 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.2 compliant implementation includes the optional architecture component or extension. The features are listed at:
  - *Architectural features added by Armv8.2* on page A2-86, which lists the original Armv8.2 architectural features.

  - *Features added to the Armv8.2 extension in later releases* on page A2-93, which lists additional Armv8.2 architectural features.
- It includes all of the Armv8.2 architectural requirements. *Additional requirements of Armv8.2* on page A2-92 lists these requirements.

For more information, see *The Armv8.2 architecture extension* on page A2-86.

The Armv8.3 architectural extension

The Armv8.3 architecture extension is an extension to Armv8.2. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.3 compliant if all of the following apply:

- It is Armv8.2 compliant.
- It includes all of the Armv8.3 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.3 compliant implementation includes the optional architecture component or extension. The features are listed at:
  - *Architectural features added by Armv8.3* on page A2-95, which lists the original Armv8.3 architectural features.
  - *Features added to the Armv8.3 extension in later releases* on page A2-97, which lists additional Armv8.3 architectural features.
- It includes all of the Armv8.3 architectural requirements. *Additional requirements of Armv8.3* on page A2-97 lists these requirements.

For more information, see *The Armv8.3 architecture extension* on page A2-95.

The Armv8.4 architectural extension

The Armv8.4 architecture extension is an extension to Armv8.3. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.4 compliant if all of the following apply:

- It is Armv8.3 compliant.
- It includes all of the Armv8.4 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.4 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.4* on page A2-100 for all of the Armv8.4 architectural features.

For more information, see *The Armv8.4 architecture extension* on page A2-100.

The Armv8.5 architectural extension

The Armv8.5 architecture extension is an extension to Armv8.4. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.5 compliant if all of the following apply:

- It is Armv8.4 compliant.
- It includes all of the Armv8.5 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.5 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.5* on page A2-105 for all of the Armv8.5 architectural features.
- It includes all of the Armv8.5 architectural requirements. *Additional requirements of Armv8.5* on page A2-107 lists these requirements.

For more information, see *The Armv8.5 architecture extension* on page A2-105.

The Armv8.6 architectural extension

The Armv8.6 architecture extension is an extension to Armv8.5. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.6 compliant if all of the following apply:

- It is Armv8.5 compliant.
- It includes all of the Armv8.6 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.6 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.6* on page A2-109 for all of the Armv8.6 architectural features.
- It includes all of the Armv8.6 architectural requirements. *Additional requirements of Armv8.6* on page A2-110 lists these requirements.

For more information, see *The Armv8.6 architecture extension* on page A2-109.

The Armv8.7 architectural extension

The Armv8.7 architecture extension is an extension to Armv8.6. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.7 compliant if all of the following apply:

- It is Armv8.6 compliant.
- It includes all of the Armv8.7 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.7 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.7* on page A2-111 for all of the Armv8.7 architectural features.
- It includes all of the Armv8.7 architectural requirements. *Additional requirements of Armv8.7* on page A2-114 lists these requirements.

For more information, see *The Armv8.7 architecture extension* on page A2-111.

The Armv8.8 architectural extension

The Armv8.8 architecture extension is an extension to Armv8.7. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.8 compliant if all of the following apply:

- It is Armv8.7 compliant.
- It includes all of the Armv8.8 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.8 compliant implementation includes the optional architecture component or extension. See *Architectural features added by Armv8.8* on page A2-115 for all of the Armv8.8 architectural features.
- It includes all of the Armv8.8 architectural requirements. *Additional requirements of Armv8.8* on page A2-118 lists these requirements.

For more information, see *The Armv8.8 architecture extension* on page A2-115.

The Statistical Profiling Extension (SPE)

SPE is an optional extension to Armv8.2. That is, SPE requires the implementation of Armv8.2.

For more information, see *The Statistical Profiling Extension (SPE)* on page A2-121.

The Scalable Vector Extension (SVE)

SVE is an optional extension to Armv8.2. That is, SVE requires the implementation of Armv8.2.

For more information, see *The Scalable Vector Extension (SVE)* on page A2-122.

The Activity Monitors Extension (AMU)

AMU is an optional extension to Armv8.4. That is, AMU requires the implementation of Armv8.4.

For more information, see *The Activity Monitors Extension (AMU)* on page A2-123.

The Memory Partitioning and Monitoring Extension (MPAM)

MPAM is an optional extension to Armv8.2. That is, MPAM requires the implementation of Armv8.2.

For more information, see *The Memory Partitioning and Monitoring (MPAM) Extension* on page A2-124.

See also *Permitted implementation of subsets of Armv8.x and Armv8.(x+1) architectural features*.

A2.1.1 Permitted implementation of subsets of Armv8.x and Armv8.(x+1) architectural features

An Armv8.*x* compliant implementation can include any arbitrary subset of the architectural features of Armv8.(*x*+1), subject only to those constraints that require that certain features be implemented together.

Unless this manual permits otherwise, an Armv8.*x* compliant implementation does not include any features of Armv8.(*x*+2) or later.

Note

The addition of Armv8.(*x*+1) features to an Armv8.*x* compliant implementation is permitted only if the implementer has a license to Armv8.(*x*+1) in addition to the license to Armv8.*x*.

A2.2 Architectural features within Armv8.0 architecture

This includes architectural features and architectural requirements that have been added to the Armv8.0 architecture since the initial release, that were not part of the original Armv8-A architecture, see:

- *Additional functionality added to Armv8.0 in later releases*.
- *Architectural requirements within Armv8.0 architecture* on page A2-79.

A2.2.1 Additional functionality added to Armv8.0 in later releases

An implementation of Armv8.0 can include any or all of the features that this section describes.

The Armv8.0 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_SB, Speculation Barrier

FEAT_SB introduces a barrier to control speculation.

This instruction is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of FEAT_SB:

- ID_AA64ISAR1_EL1.SB.
- ID_ISAR6_EL1.SB.
- ID_ISAR6.SB.

For more information, see:

- *Speculation Barrier (SB)* on page B2-173.
- *Barriers and CLREX instructions* on page C3-247.
- *Speculation Barrier (SB)* on page E2-6867.
- *Miscellaneous instructions* on page F2-6957.

FEAT_SSBS, FEAT_SBSS2, Speculative Store Bypass Safe

FEAT_SSBS allows software to indicate whether hardware is permitted to load or store speculatively in a manner that could give rise to a cache timing side channel, which in turn could be used to derive an address from values loaded to a register from memory.

FEAT_SSBS2 provides controls for the MSR and MRS instructions to read and write the PSTATE.SSBS field.

FEAT_SSBS is supported in both AArch64 and AArch32 states. FEAT_SSBS2 is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_SSBS and FEAT_SSBS2:

- ID_AA64PFR1_EL1.SSBS.
- ID_PFR2_EL1.SSBS.
- ID_PFR2.SSBS.

For more information, see:

- *Speculative Store Bypass Safe (SSBS)* on page B2-170.
- *Speculative Store Bypass Safe (SSBS)* on page E2-6863.

FEAT_CSV2 and FEAT_CSV2_2, Cache Speculation Variant 2

FEAT_CSV2 adds a mechanism to identify if hardware cannot disclose information about whether branch targets trained in one hardware described context can control speculative execution in a different hardware described context.

FEAT_CSV2_2 adds the SCXTNUM_ELx registers, which provide a number that can be used to separate out different context numbers within their respective Exception levels for the purpose of protecting against side-channels using branch prediction and similar resources.

FEAT_CSV2 is supported in both AArch64 and AArch32 states.

FEAT_CSV2_2 is supported in AArch64 state only.

FEAT_CSV2 is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

FEAT_CSV2_2 is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_CSV2:

- ID_AA64PFR0_EL1.CSV2.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

The ID_AA64PFR0_EL1.CSV2 field identifies the presence of FEAT_CSV2_2.

For more information, see:

- *Restrictions on the effects of speculation* on page B2-169.
- *Restrictions on the effects of speculation* on page E2-6863.

FEAT_CSV2_1p1 and FEAT_CSV2_1p2, Cache Speculation Variant 2

For each of these features, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p1 does not support the SCXTNUM_ELx registers, and the contexts do not include the SCXTNUM_ELx register contexts.

FEAT_CSV2_1p2 adds the SCXTNUM_ELx registers, but the contexts do not include the SCXTNUM_ELx register contexts.

These features are supported in AArch64 state only.

These features are OPTIONAL in Armv8.0 implementations.

The ID_AA64PFR1_EL1.CSV2_frac field identifies the presence of FEAT_CSV2_1p1 and FEAT_CSV2_1p2.

For more information, see:

- *Restrictions on the effects of speculation* on page B2-169.
- *Restrictions on the effects of speculation* on page E2-6863.

FEAT_CSV3, Cache Speculation Variant 3

FEAT_CSV3 adds a mechanism to identify if hardware cannot disclose information about whether data loaded under speculation with a permission or domain fault can be used to form an address, generate condition codes, or generate SVE predicate values, to be used by instructions newer than the load in the speculative sequence.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

This feature is mandatory when FEAT_E0PD is implemented.

The following fields identify the presence of FEAT_CSV3:

- ID_AA64PFR0_EL1.CSV3.
- ID_PFR2_EL1.CSV3.
- ID_PFR2.CSV3.

FEAT_SPECRES, Speculation restriction instructions

FEAT_SPECRES adds the CFP RCTX, CPP RCTX, DVP RCTX, CFPRCTX, CPPRCTX, and DVPRCTX System instructions. These instructions prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of FEAT_SPECRES:

- ID_AA64ISAR1_EL1.SPECRES.
- ID_ISAR6_EL1.SPECRES.
- ID_ISAR6.SPECRES.

For more information, see:

- *Prediction restriction instructions* on page C5-646.
- *Execution, data prediction and prefetching restriction System instructions* on page D4-4758.
- *Execution and data prediction restriction System instructions* on page G4-8818.

FEAT_CP15SDISABLE2, CP15SDISABLE2

FEAT_CP15SDISABLE2 provides an implementation-defined mechanism, the **CP15SDISABLE2** signal, which when asserted HIGH prevents writes to a set of Secure CP15 registers. This signal is analogous to the existing **CP15SDISABLE** signal.

This feature is supported only when EL3 is executing in AArch32 state.

This feature is OPTIONAL in Armv8.0 implementations.

For more information, see *The CP15SDISABLE and CP15SDISABLE2 input signals* on page G5-8969.

FEAT_DoubleLock, Double Lock

FEAT_DoubleLock is the mnemonic used for the OS Double Lock.

If FEAT_DoPD is not implemented and FEAT_Debugv8p2 is implemented, this feature is OPTIONAL.

If FEAT_DoPD is not implemented and FEAT_Debugv8p2 is not implemented, this feature is mandatory.

If FEAT_DoPD is implemented, this feature is not implemented.

Note

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

The ID_AA64DFR0_EL1.DoubleLock field identifies that the OS Double Lock has been implemented.

FEAT_DGH, Data Gathering Hint

FEAT_DGH adds the Data Gathering Hint instruction to the hint space.

This instruction is added to the A64 instruction set only.

This feature is OPTIONAL in Armv8.0 implementations.

The ID_AA64ISAR1_EL1.DGH field identifies the presence of FEAT_DGH.

For more information, see *Hint instructions* on page C3-247.

FEAT_ETS, Enhanced Translation Synchronization

FEAT_ETS adds support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.7 implementations.

The following fields identify the presence of FEAT_ETS:

- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
- ID_MMFR5.ETS.

For more information, see:

- *Ordering of memory accesses from translation table walks* on page D5-4802.
- *Ordering of translation table walks* on page E2-6871.

FEAT_nTLBPA, Intermediate caching of translation table walks

FEAT_nTLBPA adds a mechanism to identify if the intermediate caching of translation table walks does not include non-coherent caches of previous valid translation table entries since the last completed TLBI applicable to the PE.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_nTLBPA:

- ID_AA64MMFR1_EL1.nTLBPA.
- ID_MMFR5_EL1.nTLBPA.
- ID_MMFR5.nTLBPA.

For more information, see:

- *General TLB maintenance requirements* on page D5-4911.
- *General TLB maintenance requirements* on page G5-8904.

FEAT_PCSRv8, PC Sample-based Profiling Extension

FEAT_PCSRv8 adds support for PC Sample-based Profiling Extension that provides coarse-grained, non-invasive profiling by an external debugger.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_PCSRv8:

- EDDEVID.PCSample.
- DBGDEVID.PCSample.
- EDDEVID1.PCSROffset.
- DBGDEVID1.PCSROffset.
- PMDEVID.PCSample.

For more information, see *About the PC Sample-based Profiling Extension* on page H7-10028.

A2.2.2 Architectural requirements within Armv8.0 architecture

The Armv8.0 architecture includes some mandatory changes, that have been added to the architecture at a later date, that are not associated with a feature. These are:

Prefetch speculation protection

When substituting a direct branch with another direct branch, or a `NOP` with a direct branch, by the modified PE, at around the time that the executing PE is executing the software being modified, prefetch speculation protection prevents the old instructions from accidentally being fetched to the executing PE. For further information on implementation of these requirements, see:

- *Ordering of instruction fetches* on page B2-168.
- *Ordering of instruction fetches* on page E2-6862.

An implementation of the Armv8.0 architecture must comply with all of the additional requirements. When combined with the mandatory architectural features that have been added to the Armv8.0 architecture, such an implementation is also called an implementation of the Armv8.0 architecture.

A2.3 The Armv8 Cryptographic Extension

The Armv8.0 Cryptographic Extension provides instructions for the acceleration of encryption and decryption, and includes the following features:

- FEAT_AES, which includes the `AESD` and `AESE` instructions.
- FEAT_PMULL, which includes the `PMULL, PMULL2` instructions.
- FEAT_SHA1, which includes the `SHA1*` instructions.
- FEAT_SHA256, which includes the `SHA256*` instructions.

From Armv8.2, an implementation of the Armv8.0 Cryptographic Extension can include either or both of:

- The AES functionality, including support for multiplication of 64-bit polynomials. The ID_AA64ISAR0_EL1.AES field indicates whether this functionality is supported.
- The SHA1 and SHA2-256 functionality. The ID_AA64ISAR0_EL1.{SHA2, SHA1} fields indicate whether this functionality is supported.

The presence of the Cryptographic Extension in an implementation is subject to export license controls. The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file.

The Cryptographic Extension also provides multiply instructions that operate on long polynomials.

The Cryptographic Extension provides this functionality in AArch64 state and AArch32 state, and an implementation that supports both AArch64 state and AArch32 state provides the same Cryptographic Extension functionality in both states.

For more information, see *The Cryptographic Extension* on page C3-326 or *The Cryptographic Extension in AArch32 state* on page F2-6974.

A2.3.1 Armv8.2 extensions to the Cryptographic Extension

Armv8.2 adds optional extensions to the Armv8 Cryptographic Extension, that provide cryptographic functionality in AArch64 state only. These optional features are:

FEAT_SHA512, Advanced SIMD SHA512 instructions

FEAT_SHA512 adds Advanced SIMD instructions that support SHA2-512 functionality.

These instructions are added to the A64 instruction set only.

Implementation of FEAT_SHA512 requires implementation of the Armv8.0 Cryptographic Extension FEAT_SHA1 and FEAT_SHA256 functionality.

The ID_AA64ISAR0_EL1.SHA2 field identifies the presence of FEAT_SHA512.

For more information, see *FEAT_SHA512, SHA2-512 functionality* on page C3-327.

FEAT_SHA3, Advanced SIMD SHA3 instructions

FEAT_SHA3 adds Advanced SIMD instructions that support SHA3 functionality.

These instructions are added to the A64 instruction set only.

Implementation of FEAT_SHA3 requires implementation of the Armv8.0 Cryptographic Extension FEAT_SHA1 and FEAT_SHA256 functionality.

The ID_AA64ISAR0_EL1.SHA3 field identifies the presence of FEAT_SHA3.

For more information, see *FEAT_SHA3, SHA3 functionality* on page C3-328.

FEAT_SM3, Advanced SIMD SM3 instructions

FEAT_SM3 adds Advanced SIMD instructions that support the Chinese cryptography algorithm SM3.

These instructions are added to the A64 instruction set only.

Implementation of FEAT_SM3 is independent of the implementation of any SHA functionality.

The ID_AA64ISAR0_EL1.SM3 field identifies the presence of FEAT_SM3.

For more information, see *FEAT_SM3, SM3 functionality* on page C3-329.

FEAT_SM4, Advanced SIMD SM4 instructions

FEAT_SM4 adds Advanced SIMD instructions that support the Chinese cryptography algorithm SM4.

Implementation of FEAT_SM4 is independent of the implementation of any SHA functionality.

These instructions are added to the A64 instruction set only.

The ID_AA64ISAR0_EL1.SM4 field identifies the presence of FEAT_SM4.

For more information, see *FEAT_SM4, SM4 functionality* on page C3-329.

A2.4 The Armv8.1 architecture extension

The Armv8.1 architecture extension adds both architectural features and architectural requirements, see:

- *Architectural features added by Armv8.1*.
- *Additional requirements of Armv8.1* on page A2-84.
- *Features added to the Armv8.1 extension in later releases* on page A2-85.
- *Features made optional in Armv8.1 implementations* on page A2-85.

A2.4.1 Architectural features added by Armv8.1

An implementation of the Armv8.1 extension must include all of the features that this section describes as mandatory. Such an implementation, when combined with the additional requirements of Armv8.1, is also called an implementation of the Armv8.1 architecture.

The Armv8.1 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_LSE, Large System Extensions

FEAT_LSE introduces a set of atomic instructions:

- Compare and Swap instructions, `CAS` and `CASP`.
- Atomic memory operation instructions, `LD<OP>` and `ST<OP>`, where `<OP>` is one of `ADD`, `CLR`, `EOR`, `SET`, `SMAX`, `SMIN`, `UMAX`, and `UMIN`.
- Swap instruction, `SWP`.

These instructions are added only to the A64 instruction set.

This feature is mandatory in Armv8.1 implementations.

Implementations of FEAT_VHE require the implementation of FEAT_LSE.

The ID_AA64ISAR0_EL1.Atomic field identifies the presence of FEAT_LSE.

For more information, see:

- *Atomic memory operations* on page C3-265.
- *Swap* on page C3-268.
- *Compare and Swap* on page C3-269.

FEAT_RDM, Advanced SIMD rounding double multiply accumulate instructions

FEAT_RDM introduces Rounding Double Multiply Add/Subtract Advanced SIMD instructions. For more information, see:

For the A64 instruction set

- *SQRDMLAH (by element)* on page C7-2673.
- *SQRDMLAH (vector)* on page C7-2676.
- *SQRDMLSH (by element)* on page C7-2679.
- *SQRDMLSH (vector)* on page C7-2682.

For the T32 and A32 instruction sets

- *VQRDMLAH* on page F6-8343.
- *VQRDMLSH* on page F6-8347.

This feature is mandatory in Armv8.1 implementations.

The following fields identify the presence of FEAT_RDM:

- ID_AA64ISAR0_EL1.RDM.
- ID_ISAR5_EL1.RDM.
- ID_ISAR5.RDM.

FEAT_LOR, Limited ordering regions

Limited ordering regions allow large systems to perform special Load-Acquire and Store-Release instructions that provide order between the memory accesses to a region of the PA map as observed by a limited set of observers.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.1 implementations.

The ID_AA64MMFR1_EL1.LO field identifies the presence of FEAT_LOR.

For more information, see:

- *Limited ordering regions* on page B2-179.

FEAT_HPDS, Hierarchical permission disables

FEAT_HPDS introduces the facility to disable the hierarchical attributes, APTable, PXNTable, and UXNTable, in the translation tables. This disable has no effect on the NSTable bit.

This feature is mandatory in Armv8.1 implementations.

This feature is added only to the VMSAv8-64 translation regimes. Armv8.2 extends this to the AArch32 translation regimes, see FEAT_AA32HPD.

The ID_AA64MMFR1_EL1.HPDS field identifies the presence of FEAT_HPDS.

FEAT_HAFDBS, Hardware management of the Access flag and dirty state

In Armv8.0, all updates to the translation tables are performed by software. From Armv8.1, for the VMSAv8-64 translation regimes only, hardware can perform updates to the translation tables in two contexts:

- Hardware management of the Access flag.
- Hardware management of dirty state, with updates to a dirty state in the translation tables.

The dirty state is introduced in Armv8.1.

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

This feature is OPTIONAL in Armv8.1 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.

The ID_AA64MMFR1_EL1.HAFDBS field identifies the presence of FEAT_HAFDBS.

For more information, see:

- *The dirty state* on page D5-4861.
- *Hardware management of the Access flag and dirty state* on page D5-4862.

FEAT_PAN, Privileged access never

FEAT_PAN adds a bit to PSTATE. When the value of this PAN state bit is 1, any privileged data access from EL1, or EL2 when HCR_EL2.E2H is 1, to a virtual memory address that is accessible to data accesses at EL0, generates a Permission fault.

This feature is mandatory in Armv8.1 implementations.

This feature is supported in both AArch64 and AArch32 states.

The following fields identify the presence of FEAT_PAN:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

For more information, see:

- *About PSTATE.PAN* on page D5-4850.
- *About the PAN bit* on page G5-8879.

FEAT_VMID16, 16-bit VMID

In an Armv8.1 implementation, when EL2 is using AArch64, the *virtual machine identifier* (*VMID*) size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

This feature is OPTIONAL in Armv8.1 implementations.

When implemented, this feature is supported only when EL2 is using AArch64.

The ID_AA64MMFR1_EL1.VMIDBits field identifies the supported VMID size.

For more information, see:

- *VMID size* on page D5-4907.

FEAT_VHE, Virtualization Host Extensions

Armv8.1 introduces the *Virtualization Host Extensions* (VHE) that provide enhanced support for Type 2 hypervisors in Non-secure state.

This feature is mandatory in Armv8.1 implementations.

An implementation that includes FEAT_VHE requires FEAT_LSE to be implemented.

The ID_AA64MMFR1_EL1.VH field identifies the presence of FEAT_VHE.

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

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.{CopSDbg, CopDbg}.

For more information, see:

- *Virtualization Host Extensions* on page D5-4882.

FEAT_PMUv3p1, PMU Extensions v3.1

Armv8.1 makes the following enhancements to the Performance Monitors Extension:

- The event number space is extended to 16 bits to allow additional IMPLEMENTATION DEFINED event types, and the reserved space for future additions to the architecturally-defined event types is extended.
- The HPMD bit is added to MDCR_EL2. This bit disables event counting at EL2.
- The STALL_FRONTEND and STALL_BACKEND events are required to be implemented. For more information, see *Required events* on page D8-5139.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Arm8.1 implementation must include FEAT_PMUv3p1.

The following fields identify the presence of FEAT_PMUv3p1:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.

A2.4.2 Additional requirements of Armv8.1

The Armv8.1 architecture includes some mandatory changes that are not associated with a feature. These are:

Changes to CRC32 instructions

All implementations of the Armv8.1 architecture are required to implement the `CRC32*` instructions. These are OPTIONAL in Armv8.0.

The following fields identify the presence of the `CRC32*` instructions:

- ID_AA64ISAR0_EL1.CRC32.
- ID_ISAR5_EL1.CRC32.
- ID_ISAR5.CRC32.

An implementation of the Armv8.1 extension must comply with all of the additional requirements. Such an implementation, when combined with the mandatory architectural features of Armv8.1, is also called an implementation of the Armv8.1 architecture.

A2.4.3 Features added to the Armv8.1 extension in later releases

FEAT_PAN3, Support for SCTLR_ELx.EPAN

FEAT_PAN3 adds a bit to SCTLR_EL1 and SCTLR_EL2, EPAN, to support using Privileged Access Never with instruction accesses for stage 1 translation regimes.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.1 implementations and mandatory in Armv8.7 implementations.

The ID_AA64MMFR1_EL1.PAN field identifies the presence of FEAT_PAN3.

For more information, see *About PSTATE.PAN* on page D5-4850.

A2.4.4 Features made OPTIONAL in Armv8.1 implementations

The feature that has been made OPTIONAL in Armv8.1 implementations is *FEAT_PAN2* on page A2-86.

A2.5 The Armv8.2 architecture extension

The Armv8.2 architecture extension adds both architectural features and architectural requirements, see:

- *Architectural features added by Armv8.2*.
- *Additional requirements of Armv8.2* on page A2-92.
- *Features added to the Armv8.2 extension in later releases* on page A2-93.
- *Features made optional in Armv8.2 implementations* on page A2-94.

The Armv8.2 architecture extension also adds functionality to the Cryptographic Extension, see *Armv8.2 extensions to the Cryptographic Extension* on page A2-80.

A2.5.1 Architectural features added by Armv8.2

An implementation of the Armv8.2 extension must include all of the features that this section describes as mandatory. Such an implementation, when combined with the additional requirements of Armv8.2, is also called an implementation of the Armv8.2 architecture.

The Armv8.2 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_ASMv8p2, Armv8.2 changes to the A64 ISA

FEAT_ASMv8p2 adds the `BFC` instruction to the A64 instruction set as an alias of `BFM`. It also requires that the `BFC` instruction and the A64 pseudo-instruction `REV64` are implemented by assemblers.

Note

- In Armv8.0 and Armv8.1, the A64 pseudo-instruction `REV64` is OPTIONAL.
- Because this feature relates to support for an instruction alias and for a pseudo-instruction, there are no corresponding feature ID register fields.

This change to the instruction set and assembler requirements is mandatory in an Armv8.2 implementation.

For more information, see:

- *BFC* on page C6-1193.
- *REV64* on page C6-1738.

FEAT_PAN2, `AT S1E1R` and `AT S1E1W` instruction variants affected by PSTATE.PAN

FEAT_PAN2 adds variants of the AArch64 `AT S1E1R` and `AT S1E1W` instructions and the AArch32 `ATS1CPR` and `ATS1CPW` instructions. These instructions factor in the PSTATE.PAN bit when determining whether or not the location will generate a Permission fault for a privileged access, as is reported in the PAR. For more information, see:

For the AArch64 System instructions

- *AT S1E1RP, Address Translate Stage 1 EL1 Read PAN* on page C5-831.
- *AT S1E1WP, Address Translate Stage 1 EL1 Write PAN* on page C5-835.

For the AArch32 System instructions

- *ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN* on page G8-9044.
- *ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN* on page G8-9046.

This feature is OPTIONAL in Armv8.1 implementations and mandatory in Armv8.2 implementations.

These instructions are added to the A64 and A32/T32 instruction sets.

The following fields identify the presence of FEAT_PAN2:

- ID_AA64MMFR1_EL1.PAN.
- ID_MMFR3_EL1.PAN.
- ID_MMFR3.PAN.

For more information, see:

- *Address translation instructions* on page D5-4830.
- *ATS1C**, Address translation stage 1, current security state* on page G5-8956.
- *Encoding and availability of the address translation instructions* on page G5-8957.

FEAT_FP16, Half-precision floating-point data processing

FEAT_FP16 supports:

- Half-precision data-processing instructions for Advanced SIMD and floating-point in both AArch64 and AArch32 states.
- The FPCR.FZ16 and FPSCR.FZ16 bits, which enables flushing of denormalized numbers to zero for half-precision data-processing instructions.

This feature is OPTIONAL in Armv8.2 implementations, unless one of the following is implemented:

- The Scalable Vector Extension (SVE).
- FEAT_FHM.

If SVE or FEAT_FHM is implemented, FEAT_FP16 is implemented. From Armv8.4, if FEAT_FHM is not implemented, FEAT_FP16 is not implemented.

When this feature is implemented it is implemented in both Advanced SIMD and floating-point, and in AArch64 and AArch32 states.

The following fields identify the presence of FEAT_FP16:

- ID_AA64PFR0_EL1.{FP, AdvSIMD}.
- MVFR1_EL1.{FPHP, SIMDHP}.
- MVFR1.{FPHP, SIMDHP}.

For more information, see:

- *Half-precision floating-point formats* on page A1-48.
- *Flushing denormalized numbers to zero* on page A1-59.
- *Modified immediate constants in A64 instructions* on page C2-240.

FEAT_DotProd, Advanced SIMD dot product instructions

FEAT_DotProd provides instructions to perform the dot product of two 32-bit vectors, accumulating the result in a third 32-bit vector. This can be performed using signed or unsigned arithmetic.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.4 implementations.

These instructions are added to the A64 and A32/T32 instruction sets.

The following fields identify the presence of FEAT_DotProd:

- ID_AA64ISAR0_EL1.DP.
- ID_ISAR6_EL1.DP.
- ID_ISAR6.DP.

For more information, see:

- *SIMD dot product* on page C3-323.
- *Advanced SIMD dot product instructions* on page F2-6971.

FEAT_FHM, Floating-point half-precision multiplication instructions

FEAT_FHM adds floating-point multiplication instructions.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is OPTIONAL in Armv8.2 implementations, and can be implemented only when FEAT_FP16 is implemented. This feature is mandatory in Armv8.4 implementations when FEAT_FP16 is implemented. This feature is not implemented in Armv8.4 implementations when FEAT_FP16 is not implemented.

The following fields identify the presence of FEAT_FHM:

- ID_AA64ISAR0_EL1.FHM.
- ID_ISAR6_EL1.FHM.

- ID_ISAR6.FHM.

For more information, see:

- *SIMD arithmetic* on page C3-310.
- *SIMD by element arithmetic* on page C3-317.
- *Advanced SIMD multiply instructions* on page F2-6970.

FEAT_LSMAOC, AArch32 Load/Store Multiple instruction atomicity and ordering controls

FEAT_LSMAOC adds controls that disable legacy behavior of AArch32 load multiple and store multiple instructions, and provide a trap of one aspect of this legacy behavior.

Implementation of FEAT_LSMAOC is OPTIONAL. When implemented it provides:

- LSMAOE fields in the SCTLR_EL1, SCTLR_EL2, HSCTLR, and SCTLR registers. These fields can have the following effects on the behavior of AArch32 load multiple and store multiple instructions:
  - An interrupt can be taken between two memory accesses made by a single load multiple or store multiple instruction.
  - The memory accesses made by a single load multiple or store multiple instruction to Device memory with the non-Reordering attribute can be reordered.
- nTLSMD fields in the SCTLR_EL1, SCTLR_EL2, HSCTLR, and SCTLR registers. These fields can cause an access to Device-nGRE, Device-nGnRE, or Device-nGnRnE memory by an AArch32 load multiple and store multiple instruction to generate an Alignment fault.

Note

Armv8.2 deprecates software dependence on the legacy behavior of AArch32 load multiple and store multiple instructions, and these fields disable this behavior.

The following fields identify the presence of FEAT_LSMAOC:

- ID_AA64MMFR2_EL1.LSM.
- ID_MMFR4_EL1.LSM.
- ID_MMFR4.LSM.

For more information, see the register field descriptions and:

- *Generation of Alignment faults by load/store multiple accesses to Device memory* on page E2-6878.
- *Multi-register loads and stores that access Device memory* on page E2-6890.
- *Taking an interrupt or other exception during a multiple-register load or store* on page G1-8647.

FEAT_UAO, Unprivileged Access Override control

Armv8.2 adds a bit to PSTATE. When the value of PSTATE.UAO is 1, and when executed at EL1 or at EL2 with HCR_EL2.{E2H, TGE} == {1, 1}, the memory accesses made by the load/store unprivileged instructions behave as if they were made by the load/store register instructions. See *Load/store unprivileged* on page C3-256 and *Load/store register* on page C3-252.

This feature is mandatory in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64MMFR2_EL1.UAO field identifies the presence of FEAT_UAO.

For more information, see *About PSTATE.UAO* on page D5-4851.

FEAT_DPB, DC CVAP instruction

FEAT_DPB introduces a mechanism to identify and manage persistent memory locations in a shared memory hierarchy, including adding the `DC CVAP` instruction.

This feature is mandatory in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64ISAR1_EL1.DPB field identifies the presence of FEAT_DPB.

For more information about FEAT_DPB, see *Memory hierarchy* on page B2-181.

FEAT_VPIPT, VMID-aware PIPT instruction cache

FEAT_VPIPT supports a instruction cache type, described as the *VMID-aware PIPT* (VPIPT) instruction cache.

Note

Armv8.2 adds VPIPT to the set of supported cache types, meaning an Armv8.2 implementation is permitted to implement VPIPT caches, but is not required to do so.

This feature is supported in both AArch64 and AArch32 states.

The CTR_EL0.L1Ip and CTR.L1Ip fields identify the presence of FEAT_VPIPT.

For more information, see:

- *VPIPT (VMID-aware PIPT) instruction caches* on page D5-4932.
- *VPIPT (VMID-aware PIPT) instruction caches* on page G5-8920.

FEAT_AA32HPD, AArch32 hierarchical permission disables

FEAT_HPDS introduced the ability to disable the hierarchical attributes, APTable, PXNTable, and UXNTable, in the VMSAv8-64 translation regimes. FEAT_AA32HPD extends this functionality to the VMSAv8-32 translation regimes when those regimes are using the Long descriptor Translation Table format.

This feature is OPTIONAL in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.

The ID_MMFR4_EL1.HPDS and ID_MMFR4.HPDS fields identify the presence of FEAT_AA32HPD.

For more information, see *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-8860.

FEAT_HPDS2, Translation table page-based hardware attributes

Armv8.2 provides a mechanism to allow operating systems or hypervisors to make up to four bits of Translation Table final-level descriptors available for IMPLEMENTATION DEFINED hardware use.

This functionality is available for all translation regimes in AArch64 state and for stages of translation in AArch32 state that use the Long descriptor Translation Table format.

FEAT_HPDS2 is OPTIONAL in Armv8.2 implementations, but implementation of FEAT_HPDS2 requires implementation of both:

- FEAT_HPDS.
- FEAT_AA32HPD, if any Exception level higher than EL0 can use AArch32.

Note

For stage 1 translations, page-based hardware attributes can be used only for a stage of translation for which the Hierarchical permission disables field has a value of 1.

The following fields identify the presence of FEAT_HPDS2:

- ID_AA64MMFR1_EL1.HPDS.
- ID_MMFR4_EL1.HPDS.
- ID_MMFR4.HPDS.

For more information, see:

- *Memory attribute fields in the VMSAv8-64 Translation Table format descriptors* on page D5-4841.
- *Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors* on page G5-8860.

FEAT_LPA, Large PA and IPA support

FEAT_LPA:

- Allows a larger *intermediate physical address* (IPA) and PA space of up to 52 bits when using the 64KB translation granule.
- Allows a level 1 block size where the block covers a 4TB address range for the 64KB translation granule if the implementation support 52 bits of PA.

This is an OPTIONAL feature in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether it is implemented.

This feature is supported in AArch64 state only.

The ID_AA64MMFR0_EL1.PARange field identifies the presence of FEAT_LPA.

For more information about FEAT_LPA, see:

- *VMSA address types and address spaces* on page D5-4769.
- *Address size configuration* on page D5-4784.
- *Extending addressing above 48 bits when using the 64KB translation granule* on page D5-4790.
- *VMSAv8-64 translation table level -1, level 0, level 1, and level 2 descriptor formats* on page D5-4834.
- *Translation table level 3 descriptor formats* on page D5-4839.

FEAT_LVA, Large VA support

FEAT_LVA supports a larger VA space for each translation table base register of up to 52 bits when using the 64KB translation granule.

This feature is supported in AArch64 state only.

This is an OPTIONAL feature in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether it is implemented.

If FEAT_LVA is implemented, then any implemented trace macrocell must be at least ETMv4.2.

The ID_AA64MMFR2_EL1.VARange field identifies the presence of FEAT_LVA.

For more information about FEAT_LVA, see:

- *VMSA address types and address spaces* on page D5-4769.
- *Address size configuration* on page D5-4784.
- *Extending addressing above 48 bits when using the 64KB translation granule* on page D5-4790.
- *VMSAv8-64 translation table level -1, level 0, level 1, and level 2 descriptor formats* on page D5-4834.
- *Translation table level 3 descriptor formats* on page D5-4839.

FEAT_TTCNP, Translation table Common not private translations

FEAT_TTCNP permits multiple PEs in the same Inner Shareable domain to use the same translation tables for a given stage of address translation.

This feature is mandatory in Armv8.2 implementations.

This facility is available for all VMSAv8-64 translation regimes and for VMSAv8-32 translation stages that use the Long descriptor Translation Table format.

The following fields identify the presence of FEAT_TTCNP:

- ID_AA64MMFR2_EL1.CnP.
- ID_MMFR4_EL1.CnP.
- ID_MMFR4.CnP.

For more information, see:

- *Common not private translations* on page D5-4906.
- *Common not private translations in VMSAv8-32* on page G5-8909.

FEAT_XNX, Translation table stage 2 Unprivileged Execute-never

FEAT_XNX extends the stage 2 translation table access permissions to provide control of whether memory is executable at EL0 independent of whether it is executable at EL1.

This feature is mandatory in Armv8.2 implementations that implement EL2.

This facility is available for stage 2 translation stages in VMSAv8-64 and VMSAv8-32.

The following fields identify the presence of FEAT_XNX:

- ID_AA64MMFR1_EL1.XNX.
- ID_MMFR4_EL1.XNX.
- ID_MMFR4.XNX.

For more information, see:

- *Access permissions for instruction execution* on page D5-4855.
- *Access permissions for instruction execution* on page G5-8880.

FEAT_Debugv8p2, Debug v8.2

FEAT_Debugv8p2 covers a selection of mandatory changes, including:

- If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, EDPRSR.{DLK,SPD,PU} is only permitted to read {UNKNOWN, 0, 0}.
- The definition of Exception Catch debug events is extended to include reset entry.
- All CONSTRAINED UNPREDICTABLE cases that generate Exception Catch debug events are removed.
- Controls are added to EDECCR to control Exception Catch debug event generation on exception return.
- All IMPLEMENTATION DEFINED control of external debug accesses to OSLAR_EL1 is removed.
- `ExternalSecureNoninvasiveDebugEnabled()` cannot override software controls of counting attributable events in Secure state.

If FEAT_Debugv8p2 is implemented, FEAT_DoubleLock is OPTIONAL.

The fields that identify the presence of FEAT_Debugv8p2 are:

- ID_AA64DFR0_EL1.DebugVer and DBGDIDR.Version.
- ID_DFR0_EL1.{CopSDbg, CopDbg} and ID_DFR0.{CopSDbg, CopDbg}.
- EDDEVARCH.ARCHID.

For more information, see:

- *Exception Catch debug event* on page H3-9963.
- *EDPRSR.{DLK, SPD, PU} and the Core power domain* on page H6-10018.
- *Interaction with EL3* on page D8-4959.
- *External access disabled* on page H8-10040.

FEAT_PCSRv8p2, PC Sample-based profiling

In Armv8.2, the control and implementation of the OPTIONAL PC Sample-based Profiling extension is moved from ED*SR Debug registers to PM*SR registers in the Performance Monitors address space. See Chapter H7 *The PC Sample-based Profiling Extension*.

The PC Sample-based Profiling Extension is an OPTIONAL feature. If it is implemented, an Arm8.2 implementation must also include FEAT_PCSRv8p2.

If Secure EL2 and PC Sample-based Profiling are both implemented, FEAT_PCSRv8p2 is mandatory.

The following fields identify the presence of FEAT_PCSRv8p2:

- EDDEVID.PCSample.
- DBGDEVID.PCSample.
- EDDEVID1.PCSROffset.
- DBGDEVID1.PCSROffset.

- PMDEVID.PCSample.

FEAT_IESB, Implicit Error Synchronization event

FEAT_IESB adds an implicit error synchronization event at exception entry and return, controlled by the added SCTLR_ELx.IESB fields. An IESB field is added to the ESR_ELx syndrome registers.

The implicit error synchronization events affect the same synchronizable asynchronous events that are synchronized by the `ESB` instruction, see *The Reliability, Availability, and Serviceability Extension* on page A2-120.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64MMFR2_EL1.IESB field identifies the presence of FEAT_IESB.

For more information, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

FEAT_F32MM, Single-precision Matrix Multiplication

FEAT_F32MM adds support for the SVE FP32 single-precision floating-point matrix multiplication variant of the `FMMLA` instruction.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is supported in AArch64 state only.

FEAT_F32MM requires FEAT_SVE.

The ID_AA64ZFR0_EL1.F32MM field identifies the presence of FEAT_F32MM.

FEAT_F64MM, Single-precision Matrix Multiplication

FEAT_F64MM adds support for the following SVE instructions:

- `FMMLA` (FP64 double-precision variant).
- `LD1ROB (scalar plus immediate)`.
- `LD1ROB (scalar plus scalar)`.
- `LD1ROD (scalar plus immediate)`.
- `LD1ROD (scalar plus scalar)`.
- `LD1ROH (scalar plus immediate)`.
- `LD1ROH (scalar plus scalar)`.
- `LD1ROW (scalar plus immediate)`.
- `LD1ROW (scalar plus scalar)`.
- `TRN1, TRN2 (vectors)` (128-bit variant).
- `UZP1, UZP2 (vectors)` (128-bit variant).
- `ZIP1, ZIP2 (vectors)` (128-bit variant).

This feature is OPTIONAL in Armv8.2 implementations.

This feature is supported in AArch64 state only.

FEAT_F64MM requires FEAT_SVE.

The ID_AA64ZFR0_EL1.F64MM field identifies the presence of FEAT_F64MM.

Extensions to the Arm Cryptographic Extensions

See the description of the FEAT_SHA512 and FEAT_SM3 features in *Armv8.2 extensions to the Cryptographic Extension* on page A2-80.

A2.5.2 Additional requirements of Armv8.2

The Armv8.2 architecture includes some mandatory changes that are not associated with a feature. These are:

Change to ACTLR2 and HCTLR2 registers

In AArch32 state, the ACTLR2 and HACTLR2 registers become mandatory.

Implementation of RAS Extension

The RAS Extension must be implemented, see *The Reliability, Availability, and Serviceability Extension* on page A2-120.

An implementation of the Armv8.2 extension must comply with all of the additional requirements. Such an implementation, when combined with the mandatory architectural features of Armv8.2, is also called an implementation of the Armv8.2 architecture.

If FEAT_PMUv3 is implemented, the feature FEAT_PMUv3p4 is OPTIONAL in Armv8.2 implementations.

A2.5.3 Features added to the Armv8.2 extension in later releases

FEAT_EVT, Enhanced Virtualization Traps

FEAT_EVT introduces additional traps for EL1 and EL0 Cache controls. These traps are independent of existing controls.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.2 implementations and is mandatory in Armv8.5.

ID_AA64MMFR2_EL1.EVT identifies the presence of the AArch64 traps controls.

ID_MMFR4_EL1.EVT and ID_MMFR4.EVT identify the presence of the AArch32 traps.

For more information, see:

- HCR_EL2.{TTLBIS, TTLBOS, TICAB, TOCU, TID4}.
- HCR2.{TTLBIS, TICAB, TOCU, TID4}.

FEAT_DPB2, DC CVADP instruction

FEAT_DPB2 allows two levels of cache clean to the Point of Persistence by:

- Redefining Point of Persistence, which changes the scope of `DC CVAP`.
- Defining a Point of Deep Persistence.
- Adding the DC CVADP System instruction.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations and is mandatory in Armv8.5 implementations.

The ID_AA64ISAR1_EL1.DPB field identifies the presence of FEAT_DPB2.

For further information, see *Terminology for Clean, Invalidate, and Clean and Invalidate instructions* on page D4-4739.

FEAT_BF16, AArch64 BFloat16 instructions

FEAT_BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch64 state. This format supports:

- The BFloat16 floating-point data type.
- Arithmetic instructions to accelerate dot products and matrix multiplications of BF16 values.
- Instructions to convert single-precision floating-point values to BF16 format.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.BF16 field identifies the presence of FEAT_BF16.

When both Advanced SIMD and SVE are implemented, the ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 fields must return the same value.

For further information, see:

- *BFloat16 floating-point format* on page A1-52.
- *Floating-point single-precision to BFloat16 conversion instruction* on page C3-305.
- *SIMD BFloat16* on page C3-324.
- *SVE BFloat16 floating-point multiply-add* on page C3-345.
- *SVE BFloat16 floating-point dot product* on page C3-345.

- *SVE BFloat16 floating-point matrix multiply* on page C3-346.
- *SVE BFloat16 floating-point convert* on page C3-347.

FEAT_AA32BF16, AArch32 BFloat16 instructions

FEAT_AA32BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch32 state. This format supports:

- The BFloat16 floating-point data type.
- Arithmetic instructions to accelerate dot products and matrix multiplications of BF16 values.
- Instructions to convert single-precision floating-point values to BF16 format.

This feature is supported in AArch32 state only.

This feature is OPTIONAL in Armv8.2 implementations.

The ID_ISAR6_EL1.BF16 and ID_ISAR6.BF16 fields identify the presence of FEAT_AA32BF16.

For further information, see:

- *BFloat16 floating-point format* on page A1-52.
- *Advanced SIMD BFloat16 instructions* on page F2-6972.
- *Floating-point data-processing* on page F3-7014.

FEAT_I8MM, AArch64 Int8 matrix multiplication instructions

FEAT_I8MM introduces integer matrix multiply-accumulate instructions and mixed sign dot product instructions.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.I8MM field identifies the presence of FEAT_I8MM.

When both Advanced SIMD and SVE are implemented, the ID_AA64ISAR1_EL1.I8MM and the ID_AA64ZFR0_EL1.I8MM fields must return the same value.

For further information, see:

- *SIMD dot product* on page C3-323.
- *SIMD integer matrix multiply-accumulate* on page C3-326.
- *SVE Integer dot product* on page C3-334.
- *SVE Integer matrix multiply operations* on page C3-334.

FEAT_AA32I8MM, AArch32 Int8 matrix multiplication instructions

FEAT_AA32I8MM introduces integer matrix multiply-accumulate instructions and mixed sign dot product instructions.

This feature is supported in AArch32 state only.

This feature is OPTIONAL in Armv8.2 implementations.

The ID_ISAR6_EL1.I8MM and ID_ISAR6.I8MM fields identify the presence of FEAT_AA32I8MM.

For further information, see:

- *Advanced SIMD dot product instructions* on page F2-6971.
- *Advanced SIMD matrix multiply instructions* on page F2-6972.

A2.5.4 Features made OPTIONAL in Armv8.2 implementations

The features that have been made OPTIONAL in Armv8.2 implementations are:

- *FEAT_FlagM* on page A2-100.
- *FEAT_LSE2* on page A2-100.
- *FEAT_LRCPC2* on page A2-100.

A2.6 The Armv8.3 architecture extension

The Armv8.3 architecture extension adds both architectural features and additional requirements, see:

- *Architectural features added by Armv8.3*.
- *Additional requirements of Armv8.3* on page A2-97.
- *Features added to the Armv8.3 extension in later releases* on page A2-97.

A2.6.1 Architectural features added by Armv8.3

An implementation of the Armv8.3 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.3 architecture.

The Armv8.3 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_FCMA, Floating-point complex number instructions

FEAT_FCMA introduces instructions for floating-point multiplication and addition of complex numbers.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is mandatory in Armv8.3 implementations.

The half-precision versions of these instructions are implemented only if FEAT_FP16 is implemented. Otherwise they are UNDEFINED.

The fields that identify the presence of FEAT_FCMA are:

- ID_AA64ISAR1_EL1.FCMA.
- ID_ISAR5_EL1.VCMA.
- ID_ISAR5.VCMA.

For more information, see:

- *SIMD complex number arithmetic* on page C3-324.
- *Advanced SIMD complex number arithmetic instructions* on page F2-6971.

FEAT_JSCVT, JavaScript conversion instructions

FEAT_JSCVT introduces instructions that perform a conversion from a double-precision floating point value to a signed 32-bit integer, with rounding to zero. For more information, see:

For the A64 instruction set

- *FJCVTZS* on page C7-2245.

For the A32/T32 instruction set

- *VJCVT* on page F6-8105.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is mandatory in Armv8.3 implementations.

The fields that identify the presence of FEAT_JSCVT are:

- ID_AA64ISAR1_EL1.JSCVT.
- ID_ISAR6_EL1.JSCVT.
- ID_ISAR6.JSCVT.

For more information, see:

- *Floating-point conversion* on page C3-305.
- *About the A64 Advanced SIMD and floating-point instructions* on page C7-2014.
- *Advanced SIMD and floating-point instructions* on page E1-6824.
- *Floating-point data-processing instructions* on page F2-6976.

FEAT_LRCPC, Load-Acquire RCpc instructions

FEAT_LRCPC introduces three instructions to support the weaker *Release Consistency processor consistent* (RCpc) model that enables the reordering of a Store-Release followed by a Load-Acquire to a different address:

- *LDAPR* on page C6-1496.
- *LDAPRB* on page C6-1498.
- *LDAPRH* on page C6-1500.

These instructions are added to the A64 instruction set.

The feature is mandatory in Armv8.3 implementations.

The ID_AA64ISAR1_EL1.LRCPC field identifies the presence of FEAT_LRCPC.

For more information, see:

- *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-177.
- *Load-Acquire/Store-Release* on page C3-257.

FEAT_NV, Nested virtualization support

FEAT_NV provides support for a Guest Hypervisor to run in Non-secure EL1 and ensures that the Guest Hypervisor is unaware that it is running at that Exception level. A Guest Hypervisor is supported regardless of the value of HCR_EL2.E2H.

This feature is supported in AArch64 state only.

The feature is OPTIONAL in Armv8.3 implementations. This feature must be implemented if FEAT_NV2 is implemented.

The ID_AA64MMFR2_EL1.NV field identifies the presence of FEAT_NV.

For more information, see *Nested virtualization* on page D5-4888.

FEAT_CCIDX, Extended cache index

FEAT_CCIDX introduces the following registers to allow caches to be described with greater numbers of sets and greater associativity:

- A 64-bit format of CCSIDR_EL1.
- CCSIDR2_EL1.
- CCSIDR2.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.3 implementations.

The following fields identify the presence of FEAT_CCIDX:

- ID_AA64MMFR2_EL1.CCIDX.
- ID_MMFR4_EL1.CCIDX.
- ID_MMFR4.CCIDX.

For more information, see:

- *Possible formats of the Cache Size Identification Register, CCSIDR_EL1* on page D4-4733.
- *Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2* on page G4-8799.

FEAT_PAuth and FEAT_EPAC, Pointer authentication and Enhanced Pointer authentication

FEAT_PAuth adds functionality that supports address authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load.

FEAT_EPAC adds functionality that permits setting the *Pointer Authentication Code* (PAC) field to 0 on performing a PAC operation on a non-canonical address.

These features are supported in AArch64 state only.

FEAT_PAuth is mandatory in Armv8.3 implementations.

FEAT_EPAC is OPTIONAL in Armv8.3 implementations.

When FEAT_PAuth is implemented, one of the following must be true:

- Exactly one of the PAC algorithms is implemented.
- If the `PACGA` instruction and other Pointer authentication instructions use different PAC algorithms, exactly two PAC algorithms are implemented.

The PAC algorithm features are:

- FEAT_PACQARMA5.
- FEAT_PACIMP.
- FEAT_PACQARMA3.

The following fields identify the presence of FEAT_PAuth:

- ID_AA64ISAR1_EL1.{GPI, GPA, API, APA}.
- ID_AA64ISAR2_EL1.{GPA3, APA3}.

The following fields identify the presence of FEAT_EPAC:

- ID_AA64ISAR1_EL1.{API, APA}.
- ID_AA64ISAR2_EL1.APA3.

For more information, see *Pointer authentication in AArch64 state* on page D5-4772.

FEAT_PACQARMA5, Pointer authentication - QARMA5 algorithm

FEAT_PACQARMA5 adds the QARMA5 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth is implemented.

The ID_AA64ISAR1_EL1.{GPA, APA} fields identify the support for FEAT_PACQARMA5.

For more information, see *Pointer authentication in AArch64 state* on page D5-4772.

FEAT_PACIMP, Pointer authentication - IMPLEMENTATION DEFINED algorithm

FEAT_PACIMP permits an IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth is implemented.

The ID_AA64ISAR1_EL1.{GPI, API} fields identify the support for FEAT_PACIMP.

For more information, see *Pointer authentication in AArch64 state* on page D5-4772.

A2.6.2 Additional requirements of Armv8.3

If FEAT_PMUv3 is implemented, FEAT_PMUv3p4 is OPTIONAL in Armv8.3 implementations.

A2.6.3 Features added to the Armv8.3 extension in later releases

FEAT_SPEv1p1, Armv8.3 Statistical Profiling Extensions

FEAT_SPEv1p1 adds an Alignment Flag in the Events packet and filtering on this event using PMSEVFR_EL1, together with support for the profiling of Scalable Vector Extension operations.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations. An Armv8.5 implementation that includes the Statistical Profiling Extension must include FEAT_SPEv1p1.

The fields in ID_AA64DFR0_EL1.PMSVer identify the presence of FEAT_SPEv1p1.

For more information, see Chapter D10 *The Statistical Profiling Extension* and Chapter D11 *Statistical Profiling Extension Sample Record Specification*.

FEAT_DoPD, Debug over Powerdown

FEAT_DoPD provides a debug programmers' model where all debug and PMU registers are in the Core power domain, all CTI registers are in the Debug power domain. Power control is provided by a CoreSight *Granular Power Requester* (GPR) component.

When the OPTIONAL powerup mechanism is implemented and this feature is implemented, the debugger makes power control requests for the Core power domain using a CoreSight Class 0x9 ROM Table block, instead of using EDRCR.COREPURQ. EDRCR.COREPURQ is not implemented. Refer to the *ARM® CoreSight Architecture Specification* for more information.

This feature is OPTIONAL in Armv8.3 implementations.

When FEAT_DoPD is implemented:

- FEAT_DoubleLock is not implemented.
- FEAT_Debugv8p2 must be implemented.
- If PC Sample-based profiling is implemented, FEAT_PCSRv8p2 must be implemented.
- The optional Software Lock is not implemented by the architecturally defined debug components in the PE Core power domain.
- If an ETMv4 PE Trace Unit is implemented, the ETM must implement:
  - ETMv4.2 or later.
  - The Unified Power Domain Model.

The fields that identify the presence of FEAT_DoPD are:

- EDDEVID.DebugPower.
- CTIDEVARCH.REVISION.

For more information, see Chapter H6 *Debug Reset and Powerdown Support*.

FEAT_PAuth2, Enhancements to pointer authentication

FEAT_PAuth2 adds enhanced pointer authentication functionality that changes the mechanism by which a PAC is added to the pointer.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations and mandatory in Armv8.6 implementations.

The fields that identify the support for FEAT_PAuth2 are:

- ID_AA64ISAR1_EL1.{API, APA}.
- ID_AA64ISAR2_EL1.APA3.

For more information, see *Pointer authentication in AArch64 state* on page D5-4772.

FEAT_FPAC and FEAT_FPACCOMBINE, Faulting on AUT* instructions and combined pointer authentication instructions

FEAT_FPAC introduces faulting on an `AUT*` instruction.

FEAT_FPACCOMBINE introduces faulting on the combined instructions that perform pointer authentication.

FEAT_FPAC is added as a further extension to FEAT_PAuth2.

FEAT_FPACCOMBINE is added as a further extension to FEAT_FPAC.

These features are supported in AArch64 state only.

FEAT_FPAC is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth2 is implemented.

FEAT_FPACCOMBINE is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_FPAC is implemented.

The fields that identify the support for FEAT_FPAC and FEAT_FPACCOMBINE are:

- ID_AA64ISAR1_EL1.{API, APA}.
- ID_AA64ISAR2_EL1.APA3.

For more information, see *Faulting on pointer authentication* on page D5-4775.

FEAT_PACQARMA3, Pointer authentication - QARMA3 algorithm

FEAT_PACQARMA3 adds the QARMA3 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth is implemented.

The ID_AA64ISAR2_EL1.{GPA3, APA3} fields identify the support for FEAT_PACQARMA3.

For more information, see *Pointer authentication in AArch64 state* on page D5-4772.

FEAT_CONSTPACFIELD, PAC algorithm enhancement

FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field, even when the top byte is not being ignored.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth2 is implemented.

The ID_AA64ISAR2_EL1.PAC_frac field identifies the support for FEAT_CONSTPACFIELD.

For more information, see *Pointer authentication in AArch64 state* on page D5-4772.

A2.7 The Armv8.4 architecture extension

The Armv8.4 architecture extension adds architectural features, see *Architectural features added by Armv8.4*. It also adds features to earlier architecture extensions, see *Features added to earlier extensions* on page A2-104.

A2.7.1 Architectural features added by Armv8.4

An implementation of the Armv8.4 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.4 architecture.

The Armv8.4 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_DIT, Data Independent Timing instructions

FEAT_DIT provides independent timing for data processing instructions with the addition of the PSTATE.DIT and CPSR.DIT fields.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory in Armv8.4 implementations.

The following fields identify the presence of FEAT_DIT:

- ID_AA64PFR0_EL1.DIT.
- ID_PFR0_EL1.DIT.
- ID_PFR0.DIT.

For more information, see:

- *About PSTATE.DIT* on page B1-145.
- *About the DIT bit* on page E1-6823.

FEAT_FlagM, Flag manipulation instructions v2

FEAT_FlagM provides instructions which manipulate the PSTATE.{N,Z,C,V} flags.

These instructions are added to the A64 instruction set only.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64ISAR0_EL1.TS field identifies the presence of FEAT_FlagM.

For more information, see *Flag manipulation instructions* on page C3-297.

FEAT_LRCPC2, Load-Acquire RCpc instructions v2

FEAT_LRCPC2 provides versions of LDAPR and STLR with a 9-bit unscaled signed immediate offset.

These instructions are added to the A64 instruction set only.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64ISAR1_EL1.LRCPC field identifies the presence of FEAT_LRCPC2.

For more information, see:

- *Changes to single-copy atomicity in Armv8.4* on page B2-153.
- *Non-exclusive Load-Acquire and Store-Release instructions* on page C3-258.
- *A64 instructions that are changed in Debug state* on page H2-9919.

FEAT_LSE2, Large System Extensions v2

FEAT_LSE2 introduces changes to single-copy atomicity requirements for loads and stores, and changes to alignment requirements for loads and stores.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.AT field identifies the presence of FEAT_LSE2.

For more information, see:

- *Requirements for single-copy atomicity* on page B2-152.
- *Alignment of data accesses* on page B2-186.

FEAT_TLBIOS, TLB invalidate instructions in Outer Shareable domain

FEAT_TLBIOS provides TLBI maintenance instructions that extend to the Outer Shareable domain.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The field ID_AA64ISAR0_EL1.TLB identifies the presence of FEAT_TLBIOS.

For more information, see:

- *TLB maintenance instruction syntax* on page D5-4915.

FEAT_TLBIRANGE, TLB invalidate range instructions

FEAT_TLBIRANGE provides TLBI maintenance instructions that apply to a range of input addresses. FEAT_TLBIRANGE being implemented implies that FEAT_TLBIOS is implemented.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The field ID_AA64ISAR0_EL1.TLB identifies the presence of FEAT_TLBIRANGE.

For more information, see:

- *TLB maintenance instruction syntax* on page D5-4915.
- *TLB range maintenance instructions* on page D5-4923.

FEAT_TTL, Translation Table Level

FEAT_TTL provides the TTL field to indicate the level of translation table walk holding the leaf entry for the address that is being invalidated. This field is provided in all TLB maintenance instructions that take a VA or an IPA argument.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The field ID_AA64MMFR2_EL1.TTL identifies the presence of FEAT_TTL.

For more information, see:

- *TLB maintenance instruction syntax* on page D5-4915.
- *TLB range maintenance instructions* on page D5-4923.

FEAT_S2FWB, Stage 2 forced Write-Back

FEAT_S2FWB reduces the requirement of additional cache maintenance instructions in systems where the data Cacheability attributes used by the Guest operating system are different from those expected by the Hypervisor.

This feature is supported in AArch64 state.

This feature is mandatory in Armv8.4 implementations that implement EL2.

The ID_AA64MMFR2_EL1.FWB field identifies the presence of FEAT_S2FWB.

For more information, see:

- *Memory region attributes* on page D5-4871.
- *The stage 2 memory region attributes, EL1&0 translation regime* on page D5-4873.

FEAT_TTST, Small translation tables

FEAT_TTST relaxes the lower limit on the size of translation tables, by increasing the maximum permitted value of the T1SZ and T0SZ fields in TCR_EL1, TCR_EL2, TCR_EL3, VTCR_EL2 and VSTCR_EL2.

This feature is supported in AArch64 state only.

This feature is mandatory if FEAT_SEL2 is implemented.

This feature is OPTIONAL if FEAT_SEL2 is not implemented.

The ID_AA64MMFR2_EL1.ST field identifies the presence of FEAT_TTST.

For more information, see:

- *Input address size* on page D5-4786.
- *Overview of the VMSAv8-64 address translation stages* on page D5-4803.

FEAT_BBM, Translation table break-before-make levels

FEAT_BBM provides support to identify the requirements of hardware to have break-before-make sequences when changing between block size for a translation.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.BBM field identifies the presence of FEAT_BBM.

For more information, see:

- *Memory attribute fields in the VMSAv8-64 Translation Table format descriptors* on page D5-4841.
- *Support levels for changing block size* on page D5-4914.

FEAT_SEL2, Secure EL2

FEAT_SEL2 permits EL2 to be implemented in Secure state. When Secure EL2 is enabled, a translation regime is introduced that follows the same format as the other Secure translation regimes.

This feature is not supported if EL2 is using AArch32.

This feature is mandatory in Armv8.4 implementations that implement both EL2 and Secure state.

The ID_AA64PFR0_EL1.SEL2 field identifies the presence of FEAT_SEL2.

For more information, see:

- *The VMSAv8-64 address translation system* on page D5-4777.

FEAT_NV2, Enhanced nested virtualization support

FEAT_NV2 supports nested virtualization by redirecting register accesses that would be trapped to EL1 and EL2 to access memory instead. The address of the memory access depends on information held in introduced register, VNCR_EL2.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.NV field identifies the presence of FEAT_NV2.

For more information, see *Enhanced support for nested virtualization* on page D5-4890.

FEAT_IDST, ID space trap handling

FEAT_IDST causes all AArch64 read accesses to the feature ID space when exceptions are generated to be reported in ESR_ELx using the EC code `0x18`.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.IDS field identifies the presence of FEAT_IDST.

FEAT_CNTSC, Generic Counter Scaling

FEAT_CNTSC adds a scaling register to the memory-mapped counter module that allows the frequency of the counter that is generated to be scaled from the basic frequency reported in the counter ID mechanisms.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.4 implementations.

The CNTID.CNTSC field identifies the presence of FEAT_CNTSC.

For more information, see:

- *CNTCR, Counter Control Register* on page I5-10550.

FEAT_Debugv8p4, Debug v8.4

FEAT_Debugv8p4 covers a selection of mandatory changes:

- The fields MDCR_EL3.{EPMAD, EDAD} control Non-secure access to the debug and PMU registers. The bus Requester is responsible for other debug authentication.
- The Software Lock is obsolete.
- Non-invasive Debug controls are relaxed.
- Secure and Non-secure views of the debug registers are enabled.

This feature is mandatory if FEAT_SEL2 is implemented.

The fields that identify the presence of FEAT_Debugv8p4 are:

- ID_AA64DFR0_EL1.DebugVer.
- DBGDIDR.Version.
- ID_DFR0_EL1.{CopSDbg, CopDbg}.
- ID_DFR0.{CopSDbg, CopDbg}.
- EDDEVARCH.ARCHID.

For more information, see:

- *Definition and constraints of a debugger in the context of external debug* on page H1-9904
- *External debug interface register access permissions* on page H8-10040

FEAT_TRF, Self-hosted Trace Extensions

FEAT_TRF adds controls of trace in a self-hosted system through System registers.

The feature provides:

- Control of Exception levels and Security states where trace generation is prohibited.
- Control of whether an offset is used for the timestamp recorded with trace information.
- A context synchronization instruction `TSB CSYNC` which can be used to prevent reordering of trace operation accesses with respect to other accesses of the same System registers.

If an ETM Architecture PE Trace Unit is implemented and the ETM PE Trace Unit includes System register access to its control registers, this feature is mandatory. If a different PE Trace Unit is implemented or the ETM PE Trace Unit does not include System register access to its control registers, this feature is OPTIONAL.

The reset state of the PE has prohibited regions controlled by the feature and not the external authentication signals. An external trace controller must override the internal controls before enabling trace, including trace from reset. This is a change from previous trace architectures and is not backwards-compatible.

The fields that identify the presence of FEAT_TRF are:

- ID_AA64DFR0_EL1.TraceFilt.
- ID_DFR0_EL1.TraceFilt.
- ID_DFR0.TraceFilt.
- EDDFR.TraceVer.
- ID_AA64DFR0_EL1.TraceVer.

For more information, see:

- Chapter D3 *AArch64 Self-hosted Trace*.
- Chapter G3 *AArch32 Self-hosted Trace*.

FEAT_PMUv3p4, PMU Extensions v3.4

FEAT_PMUv3p4 introduces the PMMIR_EL1 and PMMIR registers.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Armv8.4 implementation must include FEAT_PMUv3p4.

The fields that identify the presence of FEAT_PMUv3p4 are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see *PMU events and event numbers* on page D8-4980.

FEAT_RASv1p1, RAS Extension v1.1

FEAT_RASv1p1 implements RAS System Architecture v1.1 and adds support for:

- Simplifications to ERR<n>STATUS.
- Additional ERR<n>MISC<m> registers.
- The OPTIONAL RAS Common Fault Injection Model Extension.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.4 implementations.

The following fields identify the complete or partial presence of FEAT_RASv1p1:

- ID_AA64PFR0_EL1.RAS.
- ID_AA64PFR1_EL1.RAS_frac.
- ID_PFR0_EL1.RAS.
- ID_PFR2_EL1.RAS_frac.
- ID_PFR0.RAS.
- ID_PFR2.RAS_frac.

For more information, see:

- *The Reliability, Availability, and Serviceability Extension* on page A2-120.
- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.*

FEAT_DoubleFault, Double Fault Extension

FEAT_DoubleFault provides two controls:

- SCR_EL3.EASE.
- SCR_EL3.NMEA.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations if EL3 is implemented and EL3 uses AArch64. Otherwise, it is not implemented.

This feature is implemented if ID_AA64PFR0_EL1.RAS >= `0b0010` and the implementation includes EL3 using AArch64.

For more information, see:

- *The Reliability, Availability, and Serviceability Extension* on page A2-120.
- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.*

A2.7.2 Features added to earlier extensions

The existing functionality of OS Double Lock is added as a feature mnemonic in Armv8.0, see *FEAT_DoubleLock* on page A2-78.

A2.8 The Armv8.5 architecture extension

The Armv8.5 architecture extension adds architectural features and additional requirements, see:

- *Architectural features added by Armv8.5*.
- *Additional requirements of Armv8.5* on page A2-107.
- *Features added to earlier extensions* on page A2-108.
- *Architectural requirements added to earlier extensions* on page A2-108.
- *Features added to the Armv8.5 extension in later releases* on page A2-108.

A2.8.1 Architectural features added by Armv8.5

An implementation of the Armv8.5 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.5 architecture.

The Armv8.5 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_FlagM2, Enhancements to flag manipulation instructions

FEAT_FlagM2 provides instructions that convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format described in *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-1142.

These instructions are added to the A64 instruction set only.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64ISAR0_EL1.TS field identifies the presence of FEAT_FlagM2.

For more information, see:

- *Flag manipulation instructions* on page C3-297.
- *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-1142.

FEAT_FRINTTS, Floating-point to integer instructions

FEAT_FRINTTS provides instructions that round a floating-point number to an integral valued floating-point number that fits in a 32-bit or 64-bit integer number range.

These instructions are added to the A64 instruction set only.

This feature requires SIMD&FP, and is mandatory in Armv8.5 implementations when SIMD&FP is implemented.

The ID_AA64ISAR1_EL1.FRINTTS identifies the presence of FEAT_FRINTTS.

For more information, see *Floating-point round to integral value* on page C3-306.

FEAT_ExS, Context synchronization and exception handling

FEAT_ExS provides a mechanism to control whether exception entry and exception return are context synchronization events. Fields in the SCTLR_ELx registers enable and disable context synchronization at exception entry and return at an Exception level.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.5 implementations.

The ID_AA64MMFR0_EL1.ExS identifies the presence of FEAT_ExS.

For more information, see:

- *SCTLR_EL1, System Control Register (EL1)* on page D13-5857, SCTLR_EL2 and SCTLR_EL3.
- *Context synchronization event* on page Glossary-11510

FEAT_GTG, Guest translation granule size

FEAT_GTG allows a hypervisor to support different granule sizes for stage 2 and stage 1 translation, and allows a nested hypervisor to determine what stage 2 granule sizes are available.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64MMFR0_EL1.{TGran16_2, TGran64_2, TGran4_2} fields identify whether each of the granule sizes is supported for stage 2 translation. The ID_AA64MMFR0_EL1.{TGran16, TGran64, TGran4} fields identify whether each of the granule sizes is supported for stage 1 translations.

For more information, see *Memory translation granule size* on page D5-4793.

FEAT_BTI, Branch Target Identification

FEAT_BTI allows memory pages to be guarded against the execution of instructions that are not the intended target of a branch. To do this, it introduces:

- The GP field, which denotes the blocks and pages in stage 1 translation tables that are guarded pages.
- The PSTATE.BTYPE field, which is used to determine whether an access to a guarded memory region will generate a Branch Target exception.
- The `BTI` instruction, which is used to guard against the execution of instructions that are not the intended target of a branch.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64PFR1_EL1.BT field identifies the presence of FEAT_BTI.

For more information, see:

- *Exception entry* on page D1-4604.
- *Synchronous exception types* on page D1-4609.
- *VMSAv8-64 translation table level -1, level 0, level 1, and level 2 descriptor formats* on page D5-4834.
- *About PSTATE.BTYPE* on page D5-4851.
- *Effect of entering Debug state on PSTATE* on page H2-9916.

FEAT_E0PD, Preventing EL0 access to halves of address maps

FEAT_E0PD prevents access at EL0 to half of the addresses in the memory map.

This feature is supported in AArch64 state only. When EL1 is using AArch64 state, this feature affects access to EL0, in either Execution state.

This feature is mandatory in Armv8.5 implementations.

Implementations that support FEAT_E0PD must also support FEAT_CSV3.

The ID_AA64MMFR2_EL1.E0PD field identifies presence of FEAT_E0PD.

For more information, see:

- *Preventing EL0 access to halves of the address map* on page D5-4853.
- TCR_EL1.{E0PD0, E0PD1}.
- TCR_EL2.{E0PD0, E0PD1}.

FEAT_RNG, Random number generator

FEAT_RNG introduces the RNDR and RNDRRS registers. Reads to these registers return a 64-bit random number. A read to RNDRRS will cause a reseeding of the random number before the generation of the random number that is returned.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.5 implementations.

The ID_AA64ISAR0_EL1.RNDR field identifies presence of FEAT_RNG.

- *Effect of random number generation instructions on Condition flags* on page C6-1142.
- Appendix K11 *Random Number Generation*.

FEAT_MTE and FEAT_MTE2, Memory Tagging Extension

FEAT_MTE and FEAT_MTE2 provide architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

These features are supported in AArch64 state only.

These features are OPTIONAL in Armv8.5 implementations.

The ID_AA64PFR1_EL1.MTE field identifies the presence of FEAT_MTE and FEAT_MTE2.

For more information, see:

- Chapter D6 *The Memory Tagging Extension*.
- Chapter B2 *The AArch64 Application Level Memory Model*.
- *PMU events and event numbers* on page D8-4980.
- Chapter D10 *The Statistical Profiling Extension*.
- Chapter H2 *Debug State*.

FEAT_PMUv3p5, PMU Extensions v3.5

FEAT_PMUv3p5 extends event counters to 64-bit event counters, and adds mechanisms to disable the cycle counter in Secure state and in EL2.

FEAT_PMUv3p5 relaxes the behavior of PMCR.{IMP, IDCODE}, and deprecates use of these fields.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Armv8.5 implementation must include FEAT_PMUv3p5.

The fields that identify the presence of FEAT_PMUv3p5 are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- *Behavior on overflow* on page D8-4964
- *Controlling the PMU counters* on page D8-4968.
- *PMU events and event numbers* on page D8-4980.

A2.8.2 Additional requirements of Armv8.5

The Armv8.5 architecture includes some mandatory changes that are not associated with a feature. These are:

Restrictions on effects of speculation

Further restrictions are placed on execution for:

- Execution prediction instructions that predict addresses or register values.
- Data loaded under speculation with a permission or domain fault.
- Any System register read under speculation to a register that is not architecturally accessible from the current Exception level.

For more information, see:

- *Restrictions on the effects of speculation* on page B2-169.
- *Restrictions on the effects of speculation* on page E2-6863.

Changes to CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1

CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1 must be implemented.

Changes to the input channel gate function

If the *Cross Trigger Matrix* (CTM) is implemented, the input channel gate function must be implemented.

Deprecation of EDPRCR.CWRR

EDPRCR.CWRR is deprecated.

Mandatory changes are also made to earlier architectural extensions, see *Architectural requirements added to earlier extensions*.

A2.8.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- *FEAT_SB* on page A2-76.
- *FEAT_SSBS* on page A2-76.
- *FEAT_CSV2* on page A2-76.
- *FEAT_CSV3* on page A2-77.
- *FEAT_SPECRES* on page A2-77.
- *FEAT_CP15SDISABLE2* on page A2-78.
- *FEAT_EVT* on page A2-93.
- *FEAT_DPB2* on page A2-93.
- *FEAT_SPEv1p1* on page A2-97.
- *FEAT_DoPD* on page A2-98.

A2.8.4 Architectural requirements added to earlier extensions

The additional architectural requirement that has been added to earlier extensions is *Prefetch speculation protection* on page A2-79.

A2.8.5 Features added to the Armv8.5 extension in later releases

FEAT_MTE3, MTE Asymmetric Fault Handling

FEAT_MTE3 introduces support for asymmetric Tag Check Fault handling.

This feature is OPTIONAL in Armv8.5 implementations.

This feature is mandatory from Armv8.7 when FEAT_MTE2 is implemented.

This feature is supported in AArch64 state.

The ID_AA64PFR1_EL1.MTE field identifies the presence of FEAT_MTE3.

For more information, see Chapter D6 *The Memory Tagging Extension*.

FEAT_RNG_TRAP, Trapping support for RNDR/RNDRRS

FEAT_RNG_TRAP introduces support for EL3 trapping of reads of the RNDR and RNDRRS registers.

This feature is supported in AArch64 state.

This feature is OPTIONAL in Armv8.5 implementations.

The ID_AA64PFR1_EL1.RNDR_trap field identifies the presence of FEAT_RNG_TRAP.

A2.9 The Armv8.6 architecture extension

The Armv8.6 architecture extension adds architectural features and additional requirements, see:

- *Architectural features added by Armv8.6*.
- *Additional requirements of Armv8.6* on page A2-110.

Features are also added to earlier architecture extensions, see *Features added to earlier extensions* on page A2-110.

A2.9.1 Architectural features added by Armv8.6

An implementation of the Armv8.6 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.6 architecture.

The Armv8.6 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_ECV, Enhanced Counter Virtualization

FEAT_ECV enhances the Generic Timer architecture.

When executing in AArch64 state or AArch32 state, FEAT_ECV provides:

- Self-synchronizing views of the virtual and physical timers in AArch64 and AArch32 state.
- The ability to scale the generation of the event stream.

When EL2 is using AArch64 state, FEAT_ECV provides:

- An optional offset between the EL1 or EL0 view of physical time, and the EL2 or EL3 view of physical time.
- Traps configurable in CNTHCTL_EL2 that trap EL0 and EL1 access to the virtual counter or timer registers, and accesses to the physical timer registers when they are accessed using an EL02 descriptor.

The optional offset to views of physical time, and the configurable traps in CNTHCTL_EL2, both apply to EL1 and EL0 whether EL1 and EL0 are in AArch64 state or AArch32 state.

This feature is mandatory in Armv8.6 implementations.

The ID_AA64MMFR0_EL1.ECV field identifies the presence of FEAT_ECV. The ID_PFR1_EL1.GenTimer and ID_PFR1.GenTimer fields identify support for self-synchronized counter views in AArch32 state.

For more information, see:

- *Self-hosted trace timestamps* on page D3-4725.
- *The profiling data* on page D10-5160.
- *The AArch64 view of the Generic Timer* on page D7-4950.
- *The AArch32 view of the Generic Timer* on page G6-8976.

FEAT_FGT, Fine Grain Traps

FEAT_FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or small groups of System registers and instructions, and traps to EL3 and EL2 of the Debug Communications Channel registers. The traps are independent of existing controls.

This feature is supported in AArch64, and when EL1 is using AArch64, EL0 accesses using AArch32 are also trapped.

This feature is mandatory in Armv8.6 implementations.

The ID_AA64MMFR0_EL1.FGT field identifies the presence of FEAT_FGT.

For more information, see *Configurable instruction controls* on page D1-4628.

FEAT_TWED, Delayed Trapping of WFE

FEAT_TWED introduces support for configurable delayed trapping of the `WFE` instruction.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.6 implementations.

The ID_AA64MMFR1_EL1.TWED field identifies the presence of FEAT_TWED.

For more information, see *The Wait for Event and Wait for Event with Timeout instructions* on page D1-4640.

FEAT_AMUv1p1, AMU Extensions v1.1

FEAT_AMUv1p1 introduces support for virtualization of Activity Monitors event counters, and introduces controls to disable access to auxiliary event counters below the highest Exception level.

This feature is supported in AArch32 state and AArch64 state, if the hypervisor is using AArch64.

This feature is OPTIONAL in Armv8.6 implementations if the OPTIONAL FEAT_AMUv1 is implemented.

The fields ID_AA64PFR0_EL1.AMU, ID_PFR0_EL1.AMU, and ID_PFR0.AMU identify the presence of FEAT_AMUv1p1.

For more information, see Chapter D9 *The Activity Monitors Extension*.

FEAT_MTPMU, Multi-threaded PMU Extensions

FEAT_MTPMU introduces controls to disable PMEVTYPER<n>_EL0.MT.

This feature requires at least one of EL2 and EL3. If neither is implemented, this feature is not implemented.

If EL2 or EL3 is implemented, the feature is OPTIONAL if FEAT_PMUv3 is implemented.

Multithreaded Armv8.6 implementations with FEAT_PMUv3 implemented must implement FEAT_MTPMU to enable any multithreaded event counting.

This feature is supported in both AArch64 and AArch32 states.

The fields ID_AA64DFR0_EL1.MTPMU and ID_DFR1.MTPMU identify the presence of FEAT_MTPMU.

For more information, see:

- *Multithreaded implementations* on page D8-4972.
- MDCR_EL3.MTPME, SDCR.MTPME, MDCR_EL2.MTPME, and HDCR.MTPME.
- *Common event numbers* on page D8-5020.

A2.9.2 Additional requirements of Armv8.6

The Armv8.6 architecture includes some mandatory changes that are not associated with a feature. These are:

Changes to the frequency of the physical counter

The frequency of CNTFRQ_EL0 is standardized to a frequency of 1GHz. This means that the system counter must be implemented at 64 bits. For more information, see:

- *The system counter* on page D7-4948.
- *The system counter* on page G6-8974.

A2.9.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- *FEAT_DGH* on page A2-78.
- *FEAT_ETS* on page A2-78.
- *FEAT_BF16* on page A2-93.
- *FEAT_AA32BF16* on page A2-94.
- *FEAT_I8MM* on page A2-94.
- *FEAT_AA32I8MM* on page A2-94.
- *FEAT_PAuth2* on page A2-98.
- *FEAT_FPAC* on page A2-98.

A2.10 The Armv8.7 architecture extension

The Armv8.7 architecture extension adds architectural features and additional requirements, see:

- *Architectural features added by Armv8.7*.
- *Additional requirements of Armv8.7* on page A2-114.

Features are also added to earlier architecture extensions, see *Features added to earlier extensions* on page A2-114.

A2.10.1 Architectural features added by Armv8.7

An implementation of the Armv8.7 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.7 architecture.

The Armv8.7 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_AFP, Alternate floating-point behavior

FEAT_AFP allows alternate behavior for specified floating-point instructions including:

- Flushing of denormalized numbers to zero can be controlled separately on inputs and outputs.
- Alternate NaN propagation rules can apply.
- Output elements for specified scalar Advanced SIMD instructions can be determined using alternate rules.
- Changes to floating-point exception generation.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.7 implementations that implement floating-point support.

The ID_AA64MMFR1_EL1.AFP field identifies the presence of FEAT_AFP.

For more information, see:

- *Flushing denormalized numbers to zero* on page A1-59.
- *NaN handling and the Default NaN* on page A1-61.
- *Rounding* on page A1-63.
- *Floating-point exceptions and exception traps* on page A1-65.

FEAT_RPRES, Increased precision of Reciprocal Estimate and Reciprocal Square Root Estimate

FEAT_RPRES allows an increase in the precision of the Reciprocal Estimate and Reciprocal Square Root Estimate from an 8-bit mantissa to a 12-bit mantissa.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.7 implementations. This feature requires implementation of FEAT_AFP.

The ID_AA64ISAR2_EL1.RPRES field identifies the presence of FEAT_RPRES.

For more information, see `RecipEstimate()` and `RecipSqrtEstimate()`.

FEAT_LS64, FEAT_LS64_V, FEAT_LS64_ACCDATA, Support for 64 byte loads/stores

FEAT_LS64 introduces support for atomic single-copy 64-byte loads and stores without return. For more information, see:

- *LD64B* on page C6-1488.
- *ST64B* on page C6-1813.

FEAT_LS64_V introduces support for atomic single-copy 64-byte stores with return. For more information, see:

- *ST64BV* on page C6-1814.

FEAT_LS64_ACCDATA introduces support for atomic single-copy 64-byte EL0 stores with return. For more information, see:

- *ST64BV0* on page C6-1816.
- *ACCDATA_EL1, Accelerator Data* on page D13-5244.

These features are supported in AArch64 state only.

FEAT_LS64 is OPTIONAL in Armv8.7 implementations.

FEAT_LS64_V is OPTIONAL in Armv8.7 implementations, and can be implemented only if FEAT_LS64 is implemented.

FEAT_LS64_ACCDATA is OPTIONAL in Armv8.7 implementations, and can be implemented only if FEAT_LS64_V is implemented.

Note

The meaning of any values being returned by the ST64BV and ST64BV0 instructions are retrospectively relaxed such that they are defined by the peripheral providing the response.

The ID_AA64ISAR1_EL1.LS64 field identifies the presence of FEAT_LS64, FEAT_LS64_V, and FEAT_LS64_ACCDATA.

For more information, see *Single-copy atomic 64-byte load/store* on page C3-267.

FEAT_WFxT, `WFE` and `WFI` instructions with timeout

FEAT_WFxT introduces `WFET` and `WFIT`. These instructions support the generation of a local timeout event to act as a wake-up event for the PE when the virtual count in CNTVCT_EL0 equals or exceeds the value supplied by the instruction for the first time. The register number that holds the timeout value for trapped `WFET` and `WFIT` instructions is reported in ESR_ELx.

These instructions are added to the A64 instruction set only.

FEAT_WFxT is mandatory in Armv8.7 implementations.

The ID_AA64ISAR2_EL1.WFxT field identifies the presence of FEAT_WFxT.

For more information, see:

- *Instructions with register argument* on page C3-246.
- *Wait for Event* on page D1-4639.
- *Wait for Interrupt mechanism* on page D1-4641.

FEAT_HCX, Support for the HCRX_EL2 register

FEAT_HCX introduces the Extended Hypervisor Configuration Register, HCRX_EL2, that provides configuration controls for virtualization in addition to those provided by HCR_EL2, including defining whether various operations are trapped to EL2.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.7 implementations.

The ID_AA64MMFR1_EL1.HCX field identifies the presence of FEAT_HCX.

For more information, see *Configurable instruction controls* on page D1-4628.

FEAT_LPA2, Larger physical address for 4KB and 16KB translation granules

FEAT_LPA2:

- Allows a larger VA space for each translation table base register of up to 52 bits when using the 4KB or 16KB translation granules.
- Allows a larger intermediate physical address (IPA) and PA space of up to 52 bits when using the 4KB or 16KB translation granules.
- Allows a level 0 block size where the block covers a 512GB address range for the 4KB translation granule if the implementation supports 52 bits of PA.
- Allows a level 1 block size where the block covers a 64GB address range for the 16KB translation granule if the implementation supports 52 bits of PA.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.7 implementations. This feature requires implementation of FEAT_LPA and FEAT_LVA.

The ID_AA64MMFR0_EL1.{TGRAN4_2, TGRAN16_2, TGRAN4, TGRAN16} fields identify the presence of FEAT_LPA2.

For more information, see:

- *VMSA address types and address spaces* on page D5-4769.
- *Address size configuration* on page D5-4784.
- *Extending addressing above 48 bits when using the 4KB or 16KB translation granule* on page D5-4791.
- *VMSAv8-64 translation table level -1, level 0, level 1, and level 2 descriptor formats* on page D5-4834.
- *Translation table level 3 descriptor formats* on page D5-4839.

FEAT_XS, XS attribute

FEAT_XS introduces the XS attribute for memory to indicate that an access could take a long time to complete. This feature provides variants of DSB instructions and TLB maintenance instructions, the completion of which does not depend on the completion of memory accesses with the XS attribute.

FEAT_XS adds:

- A mechanism to define the XS attribute for memory.
- An optional nXS variant to the AArch64 `DSB` instruction and optional nXS qualifier to each AArch64 `TLBI` instruction to handle memory accesses with the XS attribute.
- The FGTnXS bit to HCRX_EL2 to determine the behavior of fine-grained traps in HFGITR_EL2 for TLB maintenance instructions with the nXS qualifier.
- The FnXS bit to HCRX_EL2 to determine the behavior of pre-existing TLB maintenance instructions in relation to the XS attribute.

This feature is supported in AArch64 state only, but the XS attribute also impacts AArch32 state execution.

This feature is mandatory in Armv8.7 implementations.

The ID_AA64ISAR1_EL1.XS field identifies the presence of FEAT_XS.

For more information, see:

- *Data Synchronization Barrier (DSB)* on page B2-175.
- *Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors* on page D5-4846.
- *The stage 1 memory region attributes* on page D5-4871.
- *Ordering and completion of TLB maintenance instructions* on page D5-4927.
- *Data Synchronization Barrier (DSB)* on page E2-6866.
- *Overview of memory region attributes for stage 1 translations* on page G5-8887.
- *Ordering and completion of TLB maintenance instructions* on page G5-8907.

FEAT_PMUv3p7, Armv8.7 PMU extensions

FEAT_PMUv3p7 adds the following features to the Performance Monitors Extension:

- PMU counters can be frozen when an event counter has an unsigned overflow.
- Event counters can be prohibited from counting events at EL3 without affecting the rest of Secure state.
- The cycle counter can be prohibited from counting cycles at EL3 without affecting the rest of Secure state.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Armv8.7 implementation must include FEAT_PMUv3p7.

The fields that identify the presence of FEAT_PMUv3p7 are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.

- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- *Controlling the PMU counters* on page D8-4968.
- *Freezing event counters* on page D8-4969.
- *Common microarchitectural events* on page D8-5031.
- *PMMIR_EL1, Performance Monitors Machine Identification Register* on page D13-6475.

FEAT_SPEv1p2, Armv8.7 SPE features

FEAT_SPEv1p2 adds the following features to the Statistical Profiling Extension, FEAT_SPE:

- Adds an inverse event filter control.
- Adds controls to freeze the PMU event counters after an SPE buffer management event occurs.
- Adds a discard mode that allows all SPE data to be discarded rather than written to memory.

This feature is mandatory from Armv8.7 when FEAT_SPE is implemented.

This feature is supported in AArch64 state.

FEAT_SPEv1p2 optionally enables support for a packet for each taken branch that provides the target address for the previous taken branch.

ID_AA64DFR0_EL1.PMSVer identifies the presence of FEAT_SPEv1p2.

If FEAT_SPEv1p2 is implemented, PMSIDR_EL1.PBT indicates support for the previous branch target packet.

For more information, see:

- *Freezing event counters* on page D8-4969.
- *Common event numbers* on page D8-5020.
- *Filtering sample records* on page D10-5158.
- *Last branch target* on page D10-5161.
- *About the Statistical Profiling Extension sample records* on page D11-5184.
- *Address packet* on page D11-5188.

A2.10.2 Additional requirements of Armv8.7

The Armv8.7 architecture includes some mandatory changes that are not associated with a feature. These are:

FEAT_ETS, Enhanced Translation Synchronization

All implementations of the Armv8.7 architecture are required to implement FEAT_ETS.

For more information, see *FEAT_ETS* on page A2-78.

A2.10.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- *FEAT_PAN3* on page A2-85.
- *FEAT_MTE3* on page A2-108

A2.11 The Armv8.8 architecture extension

The Armv8.8 architecture extension adds architectural features and additional requirements, see:

- *Architectural features added by Armv8.8*.
- *Additional requirements of Armv8.8* on page A2-118.

Features are also added to earlier architecture extensions, see *Features added to earlier extensions* on page A2-118.

A2.11.1 Architectural features added by Armv8.8

An implementation of the Armv8.8 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.8 architecture.

The Armv8.8 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_MOPS, Standardization of memory operations

FEAT_MOPS provides instructions that perform a memory copy or memory set, and adds Memory Copy and Memory Set exceptions.

FEAT_MOPS also adds the HCRX_EL2.{MSCEn, MCE2}, SCTLR_EL1.MSCEn, and SCTLR_EL2.MSCEn control bits.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64ISAR2_EL1.MOPS field identifies the presence of FEAT_MOPS.

For more information, see:

- *Memory Copy and Memory Set exceptions* on page D1-4617.

FEAT_HBC, Hinted conditional branches

FEAT_HBC provides the `BC.cond` instruction to give a conditional branch with a hint to branch prediction logic that this branch will consistently and is highly unlikely to change direction.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64ISAR2_EL1.BC field identifies the presence of FEAT_HBC.

For more information, see:

- *Conditional branch* on page C3-244.

FEAT_NMI, Non-maskable Interrupts

FEAT_NMI provides a mechanism to support *non-maskable interrupts* (NMI) and *less-masked interrupts* (LMI). In addition to legacy behavior, the feature includes the following:

- A mode for supporting an LMI interrupt mask that is distinct from PSTATE.{I, F}.
- A mode for supporting a limited NMI, where the value when PSTATE.SP is 1 is taken as an interrupt mask for all interrupts targeting that Exception level, and where the LMI interrupt mask can also be used.

FEAT_NMI adds:

- The `AllIntMask` variable.
- An optional Superpriority attribute to denote virtual and physical IRQ and FIQ interrupts as non-maskable.
- The SCTLR_ELx.{NMI, SPINTMASK} control bits.
- The PSTATE.ALLINT bit and associated instructions.
- The HCRX_EL2.TALLINT bit to enable trapping of `ALLINT` instructions at EL1.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64PFR1_EL1.NMI field identifies the presence of FEAT_NMI.

For more information, see:

- *Asynchronous exception types* on page D1-4619.
- *Virtual interrupts* on page D1-4620.
- *PSTATE fields that are meaningful in AArch64 state* on page D1-4633.
- *WFE wake-up events in AArch64 state* on page D1-4640.

FEAT_TIDCP1, EL0 use of IMPLEMENTATION DEFINED functionality

FEAT_TIDCP1 adds a control at EL1 and EL2 to enable trapping of EL0 accesses to registers that might control IMPLEMENTATION DEFINED functions.

This feature adds controls only in AArch64 state, and controls IMPLEMENTATION DEFINED execution at EL0 in both AArch32 and AArch64 states.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64MMFR1_EL1.TIDCP1 field identifies the presence of FEAT_TIDCP1.

For more information, see *Prioritization of Synchronous exceptions taken to AArch64 state* on page D1-4613.

FEAT_CMOW, Control for cache maintenance permission

FEAT_CMOW introduces support for cache maintenance instructions that controls whether:

- Cache maintenance instructions executed at EL0 require stage 1 read and write permission to prevent the instructions from generating a Permission fault.
- Cache maintenance instructions executed at EL1 or EL0 require stage 2 read and write permission to prevent the instructions from generating a Permission fault.

This feature is supported in AArch64 state only, but also impacts AArch32 instructions.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64MMFR1_EL1.CMOW field identifies the presence of FEAT_CMOW.

For more information, see:

- *A64 Cache maintenance instructions* on page D4-4742.
- *Permission fault* on page D5-4896.

FEAT_PMUv3p8, Armv8.8 PMU extensions

FEAT_PMUv3p8 adds the following features to the Performance Monitors Extension:

- The Common event number space is extended to include the ranges `0x0040-0x00BF` and `0x4040-0x40BF`.
- For an event counter *n*, if any reserved or unimplemented PMU event number is written to PMEVTYPER<n>.evtCount, the event counter *n* does not count, and a read of PMEVTYPER<n>.evtCount returns the value written.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Armv8.8 implementation must include FEAT_PMUv3p8.

The fields that identify the presence of FEAT_PMUv3p8 are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:

- *The PMU event number space and common events* on page D8-4989.
- *Common event numbers* on page D8-5020.
- *PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page D13-6461.
- *PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30* on page G8-9684.

FEAT_HPMN0, Setting of MDCR_EL2.HPMN to zero

FEAT_HPMN0 permits a hypervisor to provide zero PMU event counters for a guest operating system by setting MDCR_EL2.HPMN to zero.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory in Armv8.8 implementations that include FEAT_PMUv3 and EL2. Otherwise, this feature is OPTIONAL in implementations that include FEAT_PMUv3, FEAT_FGT, and EL2.

The fields that identify the presence of FEAT_HPMN0 are:

- ID_AA64DFR0_EL1.HPMN0.
- ID_DFR1_EL1.HPMN0.
- ID_DFR1.HPMN0.

For more information, see:

- *Interaction with EL2* on page D8-4959.
- *Controlling the PMU counters* on page D8-4968.
- *The Performance Monitors Extension* on page K1-11227.
- *The Performance Monitors Extension* on page K1-11244.

FEAT_PMUv3_TH, Event counting threshold

FEAT_PMUv3_TH adds threshold condition controls to each PMEVTYPER<n>_EL0 register. This feature permits the counter to count only when PMEVTYPER<n>.{MT, evtCount} describes an event whose count meets a specified threshold condition.

This feature is supported in both AArch64 and AArch32 states. The threshold condition controls are only accessible in AArch64 state. However, threshold conditions still apply in AArch32 state.

This feature is OPTIONAL in Armv8.8 implementations.

This feature requires FEAT_PMUv3 to be implemented, and AArch64 state to be supported.

The fields that identify the presence of FEAT_PMUv3_TH are:

- PMMIR_EL1.THWIDTH.
- PMMIR.THWIDTH.
- If the external debug interface to the PMU registers is implemented, PMMIR.THWIDTH.

For more information, see *Event counting threshold* on page D8-4976.

FEAT_SPEv1p3, Armv8.8 Statistical Profiling Extensions

FEAT_SPEv1p3 adds the following features to the Statistical Profiling Extension:

- Support for sampling Tag operations.
- Support for sampling Memory Copy and Set operations.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory from Armv8.8 when FEAT_SPE is implemented.

The ID_AA64DFR0_EL1.PMSVer field identifies the presence of FEAT_SPEv1p3.

For more information, see:

- *Additional information for each profiled memory access operation* on page D10-5162.
- *About the Statistical Profiling Extension sample records* on page D11-5184.
- *Address packet* on page D11-5188.

FEAT_Debugv8p8, Debug v8.8

FEAT_Debugv8p8 adds support to allow an asynchronous exception to be taken after an exception generates an Exception Catch debug event, but before the PE halts.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory in Armv8.8 implementations.

The fields that identify the presence of FEAT_Debugv8p8 are:

- ID_AA64DFR0_EL1.DebugVer,

- ID_DFR0.CopDbg.
- DBGDIDR.Version.
- EDDEVARCH.ARCHID.

For more information, see *Exception Catch debug event* on page H3-9963.

A2.11.2 Additional requirements of Armv8.8

The Armv8.8 architecture includes the following mandatory changes that are associated with FEAT_SPE:

Access Flag and dirty state management by SPE

Removal of the option for the Access Flag and dirty state to behave as if always disabled for accesses made by the SPU. For more information, see *Hardware management of dirty state and the Access flag by the Statistical Profiling Extension* on page D10-5179.

External abort handling by SPE

External aborts on writes made by the SPU must generate SError interrupt exceptions. For more information, see *External aborts* on page D10-5179.

The Armv8.8 architecture includes the following OPTIONAL change that is associated with FEAT_PMUv3:

64-bit external PMU programmers' model extension

The 64-bit external PMU programmers' model extension extends the Performance Monitors registers to 64 bits, except the 32-bit CoreSight management registers.

This extension is supported in both AArch64 and AArch32 states.

This extension is OPTIONAL when the external debug interface to the Performance Monitors is implemented.

This extension requires the external debug interface to the Performance Monitors to be implemented, and AArch64 state to be supported.

The PMDEVARCH.ARCHPART field indicates the presence of the 64-bit external PMU programmers' model extension.

For more information, see Chapter I3 *Recommended External Interface to the Performance Monitors*.

A2.11.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- *FEAT_PACQARMA3* on page A2-99.
- *FEAT_CONSTPACFIELD* on page A2-99.
- *FEAT_RNG_TRAP* on page A2-108.

A2.12 The Performance Monitors Extension

The Performance Monitors Extension, FEAT_PMUv3, is an OPTIONAL extension but Arm strongly recommends that Armv8-A implementations include version 3 of the Performance Monitors Extension.

ID_AA64DFR0_EL1.PMUVer indicates whether the Performance Monitors Extension is implemented.

For more information, see Chapter D8 *The Performance Monitors Extension*.

Armv8.1 introduces the following architectural feature to the Performance Monitors Extension:

- FEAT_PMUv3p1.

Armv8.4 introduces the following architectural feature to the Performance Monitors Extension:

- FEAT_PMUv3p4.

Armv8.5 introduces the following architectural feature to the Performance Monitors Extension:

- FEAT_PMUv3p5.

Armv8.6 introduces the following architectural feature to the Performance Monitors Extension:

- FEAT_MTPMU.

Armv8.7 introduces the following architectural feature to the Performance Monitors Extension:

- FEAT_PMUv3p7.

Armv8.8 introduces the following architectural features to the Performance Monitors Extension:

- FEAT_PMUv3p8.
- FEAT_HPMN0.
- FEAT_PMUv3_TH.

A2.13 The Reliability, Availability, and Serviceability Extension

The RAS Extension, FEAT_RAS, is a mandatory extension to the Armv8.2 architecture, and an OPTIONAL extension to the Armv8.0 and the Armv8.1 architectures.

The RAS Extension improves the dependability of a system by providing:

- Reliability, that is, the continuity of correct service.
- Availability, that is, the readiness for correct service.
- Serviceability, that is, the ability to undergo modifications and repairs.

ID_AA64PFR0_EL1.RAS in AArch64 state, and ID_PFR0.RAS in AArch32 state, indicate whether the RAS Extension is implemented.

The RAS Extension introduces a barrier instruction, the Error Synchronization Barrier (ESB), to the A32, T32, and A64 instruction sets.

System registers introduced by the RAS Extension are described in:

- For AArch64, *RAS registers* on page D13-6641.
- For AArch32, *RAS registers* on page G8-9759.

In addition, the RAS Extension introduces a number of memory-mapped registers. These are described in the *Arm® Architecture Reference Manual Supplement: Reliability, Availability, and Serviceability (RAS), for Armv8-A*.

Armv8.2 introduces the following architectural features to the RAS Extension:

- FEAT_IESB.

Armv8.4 introduces the following architectural features to the RAS Extension:

- FEAT_RASv1p1.
- FEAT_DoubleFault.

A2.14 The Statistical Profiling Extension (SPE)

The Statistical Profiling Extension, FEAT_SPE, is an OPTIONAL extension introduced by the Armv8.2 architecture. Implementation of the Statistical Profiling Extension requires implementation of at least Armv8.1 of the Armv8-A architecture profile. The Statistical Profiling Extension is supported only in AArch64 state.

The Statistical Profiling Extension provides a non-invasive method of sampling software and hardware using randomized sampling of either architectural instructions, as defined by the instruction set architecture, or by microarchitectural operations.

ID_AA64DFR0_EL1.PMSVer indicates whether the Statistical Profiling Extension is implemented.

For more information, see Chapter D10 *The Statistical Profiling Extension*.

Armv8.3 introduces the following architectural feature to the SPE:

- FEAT_SPEv1p1.

Armv8.7 introduces the following architectural feature to the SPE:

- FEAT_SPEv1p2.

Armv8.8 introduces the following architectural feature to the SPE:

- FEAT_SPEv1p3.

A2.15 The Scalable Vector Extension (SVE)

The Scalable Vector Extension, FEAT_SVE, is an OPTIONAL extension introduced by the Armv8.2 architecture. SVE is supported in AArch64 state only.

The Scalable Vector Extension includes the following functionality:

- Configurable vector length with scalable vector lengths from 128 bits up to 2048 bits.
- Predication using scalable predicate registers from 16 bits up to 256 bits.
- Instructions that operate on scalable size vectors and predicates.
- Gather-load and scatter-store.
- Software-managed speculative vectorization.
- System registers and fields to configure the *SVE* vector length and traps.

ID_AA64PFR0_EL1.SVE indicates whether the Scalable Vector Extension is implemented.

The Scalable Vector Extension affects some AArch64 System registers, and those register changes are identified as SVE features. SVE also introduces AArch64 System registers.

The Scalable Vector Extension complements the AArch64 Advanced SIMD and floating-point functionality. SVE does not replace the AArch64 Advanced SIMD and floating-point functionality.

Implementation of FEAT_SVE requires implementation of FEAT_FCMA and FEAT_FP16.

A2.16 The Activity Monitors Extension (AMU)

The Activity Monitors Extension is an OPTIONAL extension introduced by the Armv8.4 architecture. AMU is supported in AArch64 and AArch32 states.

The Activity Monitors Extension implements version 1 of the Activity Monitors architecture, FEAT_AMUv1, which provides a function similar to a subset of the existing Performance Monitors Extension functionality, intended for system management use rather than debugging and profiling.

The Activity Monitors Extension implements a System register interface to the Activity Monitors registers, and supports an optional external memory-mapped interface.

The fields that identify the presence of the Activity Monitors Extension are:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see Chapter D9 *The Activity Monitors Extension*.

Armv8.6 introduces the following architectural feature to the Activity Monitors Extension:

- FEAT_AMUv1p1.

A2.17 The Memory Partitioning and Monitoring (MPAM) Extension

The MPAM Extension, FEAT_MPAM, is an OPTIONAL extension introduced by the Armv8.4 architecture and requires implementation of at least Armv8.2 of the Armv8-A architecture profile. MPAM is supported in AArch64 state only.

The MPAM Extension provides a framework for memory-system component controls that partition one or more of the performance resources of the component.

The fields that identify the presence of the MPAM Extension are:

- ID_AA64PFR0_EL1.MPAM.
- EDPFR.MPAM.

For more information, see *ARM® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for ARMv8-A*.

Chapter A3
Armv9 Architecture Extensions

This chapter introduces the Armv9 architecture versions and extensions. It contains the following sections:

A3.1 Armv9-A architecture extensions

An implementation of the Armv9-A architecture must include all of the extensions that this section describes as mandatory. Such an implementation is also called an implementation of the Armv9-A architecture.

The AArch32 state might optionally be implemented at EL0. The AArch32 state is not implemented at EL1, EL2, and EL3.

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

An implementation of the Armv9-A architecture cannot include an ETM.

The Armv9 architecture adds the following features, which are identified by the architectural feature name and a short description of the feature:

FEAT_SVE2, Scalable Vector Extension version 2

The *Scalable Vector Extension version 2* (SVE2) is a superset of SVE that incorporates functionality similar to Advanced SIMD, and other enhancements. In this Manual, unless stated otherwise, when SVE is used, the behavior also applies to SVE2.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

This feature requires FEAT_SVE.

The following fields indicate the presence of FEAT_SVE2:

- ID_AA64PFR0_EL1.SVE.
- ID_AA64ZFR0_EL1.SVEver.

Although FEAT_SVE2 is OPTIONAL, standard Armv9-A software platforms support FEAT_SVE2.

For more information, see:

- *The Scalable Vector Extension (SVE)* on page A2-122.
- *Data processing - SVE2* on page C3-360.

FEAT_SVE_AES, Scalable Vector AES instructions

FEAT_SVE_AES provides the following SVE AES cryptographic instructions:

- `AESD`.
- `AESE`.
- `AESIMC`.
- `AESMC`.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_AES requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.AES field identifies support for FEAT_SVE_AES.

FEAT_SVE_BitPerm, Scalable Vector Bit Permutes instructions

FEAT_SVE_BitPerm provides the following SVE bit permute instructions:

- `BEXT`.
- `BDEP`.
- `BGRP`.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_BitPerm requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.BitPerm field identifies support for FEAT_SVE_BitPerm.

FEAT_SVE_PMULL128, Scalable Vector PMULL instructions

FEAT_SVE_PMULL128 provides the following SVE 128-bit polynomial multiply instructions:

- `PMULLB`.
- `PMULLT`.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_PMULL128 requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.AES field identifies support for FEAT_SVE_PMULL128.

FEAT_SVE_SHA3, Scalable Vector SHA3 instructions

FEAT_SVE_SHA3 provides the following SVE SHA3 cryptographic instruction:

- `RAX1`.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_SHA3 requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.SHA3 field identifies support for FEAT_SVE_SHA3.

FEAT_SVE_SM4, Scalable Vector SM4 instructions

FEAT_SVE_SM4 provides the following SVE SM4 cryptographic instructions:

- `SM4E`.
- `SM4EKEY`.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_SM4 requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.SM4 field identifies support for FEAT_SVE_SM4.

FEAT_ETE, Embedded Trace Extension

FEAT_ETE provides a trace unit that records details about software control flow running on a PE, which can be used to aid debugging or optimizing. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

This feature is OPTIONAL in an Armv9.0 implementation.

This feature requires FEAT_TRBE and FEAT_TRF.

The ID_AA64DFR0_EL1.TraceVer field identifies the presence of FEAT_ETE.

For more information on FEAT_ETE, see *Arm® Architecture Reference Manual Supplement Armv9, for Arm-v9-A architecture profile* (ARM DDI 0608).

FEAT_TRBE, Trace Buffer Extension

FEAT_TRBE enables support for a Trace Buffer Unit within a PE. When the Trace Buffer Unit is enabled, program-flow trace generated by a trace unit is written directly to memory by the Trace Buffer Unit, rather than routing trace data to a trace sink.

This feature is supported in AArch64 and AArch32 states.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_TRBE requires FEAT_ETE and FEAT_TRF.

The ID_AA64DFR0_EL1.TraceBuffer field identifies support for FEAT_TRBE.

For more information on FEAT_TRBE, see *Arm® Architecture Reference Manual Supplement Armv9, for Arm-v9-A architecture profile* (ARM DDI 0608).

FEAT_ETEv1p1, Embedded Trace Extension

FEAT_ETEv1p1 extends FEAT_ETE to support FEAT_FGT.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

This feature is OPTIONAL in an Armv9.1 implementation.

This feature requires FEAT_TRBE and FEAT_TRF.

The TRCDEVARCH.REVISION field identifies the presence of FEAT_ETEv1p1.

For more information on FEAT_ETEv1p1, see *Arm® Architecture Reference Manual Supplement Armv9, for Arm-v9-A architecture profile* (ARM DDI 0608).

FEAT_BRBE, Branch Record Buffer Extension

FEAT_BRBE provides a Branch record buffer for capturing control path history.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.2 implementation.

The ID_AA64DFR0_EL1.BRBE field identifies the presence of FEAT_BRBE.

For more information on FEAT_BRBE, see *Arm® Architecture Reference Manual Supplement Armv9, for Arm-v9-A architecture profile* (ARM DDI 0608).

FEAT_ETEv1p2, Embedded Trace Extension

FEAT_ETEv1p2 extends FEAT_ETE to support FEAT_RME.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

This feature is OPTIONAL in an Armv9.2 implementation.

This feature requires FEAT_RME and FEAT_ETEv1p1.

The TRCDEVARCH.REVISION field identifies the presence of FEAT_ETEv1p2.

For more information on FEAT_ETEv1p2, see *Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* (ARM DDI 0615).

FEAT_BRBEv1p1, Branch Record Buffer Extension

FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.3 implementation.

This feature requires FEAT_BRBE.

The ID_AA64DFR0_EL1.BRBE field identifies the presence of FEAT_BRBEv1p1.

For more information on FEAT_BRBEv1p1, see *Arm® Architecture Reference Manual Supplement Armv9, for Arm-v9-A architecture profile* (ARM DDI 0608).

FEAT_RME, Realm Management Extension

FEAT_RME is one component of the Arm Confidential Compute Architecture (Arm CCA). Together with the other components of the Arm CCA, RME enables support for dynamic, attestable, and trusted execution environments (Realms) to be run on an Arm PE.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.2 implementation.

FEAT_RME requires FEAT_RNG or FEAT_RNG_TRAP.

If the Trace Architecture is implemented, a PE that implements FEAT_RME also implements FEAT_ETEv1p2.

For more information, see *Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* (ARM DDI 0615).

FEAT_SME, Scalable Matrix Extension

FEAT_SME introduces two AArch64 execution modes that can be enabled and disabled by application software:

- In ZA mode, scalable, two-dimensional, architectural ZA tile storage becomes available and instructions are defined to load, store, extract, insert, and clear rows and columns of the ZA tiles.
- In Streaming SVE mode, the Effective SVE vector length changes to match the Effective ZA tile width, support for a substantial subset of the SVE2 instruction set is available, and, when ZA mode is also enabled, instructions are defined that accumulate the matrix outer product of two SVE vectors into a ZA tile.

This feature is supported in AArch64 state only.

This feature is OPTIONAL from Armv9.2.

The ID_AA64PFR1_EL1.SME field identifies the presence of FEAT_SME.

For more information, see *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A* (ARM DDI 0616).

FEAT_SME_FA64, Full Streaming SVE mode instructions

FEAT_SME_FA64 supports the full A64 instruction set in Streaming SVE mode.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.2 implementation.

FEAT_SME_FA64 requires FEAT_SME.

The ID_AA64SMFR0_EL1.FA64 field identifies support for FEAT_SME_FA64.

For more information, see *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A* (ARM DDI 0616).

FEAT_EBF16, AArch64 Extended BFloat16 instructions

FEAT_EBF16 supports the Extended BFloat16 mode.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.2 implementation.

FEAT_EBF16 requires FEAT_SME.

The ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 fields identify the presence of FEAT_EBF16, and must return the same value.

For more information, see *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A* (ARM DDI 0616).

FEAT_SME_F64F64, Double-precision floating-point outer product instructions

FEAT_SME_F64F64 indicates SME support for instructions that accumulate into FP64 double-precision floating-point elements in the ZA array.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.2 implementation.

FEAT_SME_F64F64 requires FEAT_SME.

The ID_AA64SMFR0_EL1.F64F64 field identifies support for FEAT_SME_F64F64.

For more information, see *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A* (ARM DDI 0616).

FEAT_SME_I16I64, 16-bit to 64-bit integer widening outer product instructions

FEAT_SME_I16I64 indicates SME support for instructions that accumulate into 64-bit integer elements in the ZA array.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.2 implementation.

FEAT_SME_I16I64 requires FEAT_SME.

The ID_AA64SMFR0_EL1.I16I64 field identifies support for FEAT_SME_I16I64.

For more information, see *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A* (ARM DDI 0616).

A3.1.1 Architectural requirements within Armv9 architecture

An Armv9.0 compliant implementation must also be Armv8.5 compliant.

An Armv9.1 compliant implementation must also be Armv8.6 and Armv9.0 compliant.

An Armv9.2 compliant implementation must also be Armv8.7 and Armv9.1 compliant.

An Armv9.3 compliant implementation must also be Armv8.8 and Armv9.2 compliant.

Part B
The AArch64 Application Level Architecture

Chapter B1
The AArch64 Application Level Programmers' Model

B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see *Exception levels* on page D1-4594.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to Chapter D1 *The AArch64 System Level Programmers' Model*.

B1.2 Registers in AArch64 Execution state

This section describes the registers and process state visible at EL0 when executing in the AArch64 state. It includes the following:

- *Registers in AArch64 state*
- *Process state, PSTATE* on page B1-140
- *System registers* on page B1-142

B1.2.1 Registers in AArch64 state

In the AArch64 application level view, an Arm processing element has:

R0-R30 31 general-purpose registers, R0 to R30. Each register can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

See the register name mapping in Figure B1-1.

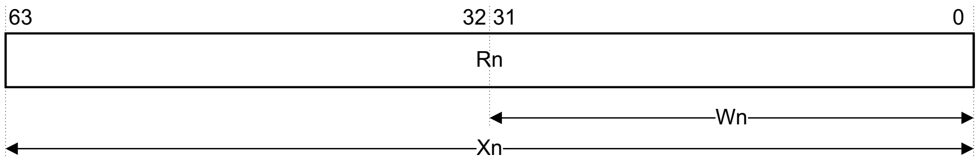


Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

Note

In instruction encodings, the value `0b11111` (31) is used to indicate the ZR (zero register). This indicates that the argument takes the value zero, but does not indicate that the ZR is implemented as a physical register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

Note

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see *PC alignment checking* on page D1-4631.

V0-V31 32 SIMD&FP registers, V0 to V31. Each register can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.
- A 128-bit vector of elements.

- A 64-bit vector of elements.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

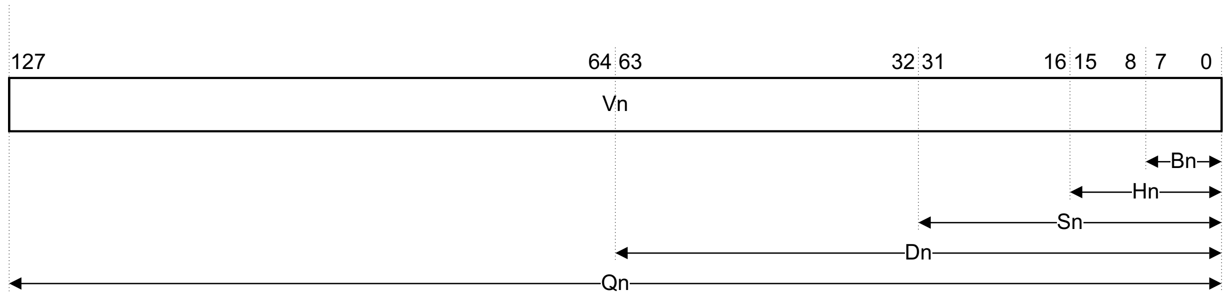


Figure B1-2 SIMD and floating-point register naming

For more information about data types and vector formats, see *Supported data types* on page A1-42.

FPCR, FPSR Two SIMD and floating-point control and status registers, FPCR and FPSR.

Z0-Z31 32 scalable vector registers, Z0 to Z31. Each register can be accessed as:

- A vector of elements with an IMPLEMENTATION DEFINED maximum length of 128 to 2048 bits.
- A fixed-length 128-bit vector of elements named V0 to V31.
- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.

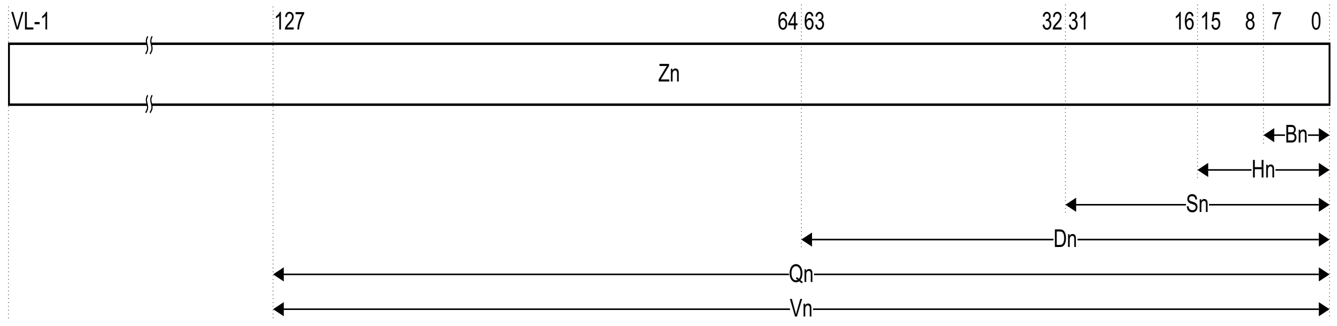


Figure B1-3 SVE register naming

For more information on SVE registers, see *SVE vector registers* on page B1-137 and *SVE writes to scalar registers* on page B1-138.

P0-P15 16 SVE scalable predicate registers, P0 to P15. See *SVE predicate registers* on page B1-139.

FFR The dedicated SVE First Fault Register that has the same size and format as the predicate registers, P0-P15. See *FFR, First Fault Register* on page B1-140.

See *Registers for instruction processing and exception handling* on page D1-4598 for more information on the registers.

Pseudocode description of registers in AArch64 state

In the pseudocode functions that access registers:

- The assignment form is used for register writes.
- The non-assignment for register reads.

The uses of the `X[]` function are:

- Reading or writing X0-X30, using n to index the required register.
- Reading the zero register ZR, accessed as `X[31]`.

Note

The pseudocode use of X[31] to represent the zero register does not indicate that hardware must implement this register.

The AArch64 `SP[]` function is used to read or write the current SP.

The AArch64 `PC[]` function is used to read the PC.

The AArch64 `V[]` function is used to read or write the Advanced SIMD and floating-point registers V0-V31, using a parameter `n` to index the required register.

The AArch64 `Vpart[]` function is used to read or write a part of one of V0-V31, using a parameter `n` to index the required register, and a parameter `part` to indicate the required part of the register, see the function description for more information.

The AArch64 `Z[]` function is used to read or write the SVE scalable vector registers Z0-Z31, using a parameter n to index the required register.

The `Z[]`, `V[]` and `Vpart[]` functions access the same underlying vector register file.

The `SP[]`, `PC[]`, `V[]`, `Vpart[]`, and `Z[]` functions are defined in Chapter J1 *Armv8 Pseudocode*.

B1.2.2 SVE vector registers

R_{FBGSJ} SVE has 32 scalable vector registers named Z0-Z31.

R_{WJNYD} All SVE scalable vector registers are the same size.

R_{KCWQB} The size of an SVE scalable vector register is an IMPLEMENTATION DEFINED multiple of 128 bits.

R_{KJSDQ} The maximum size of an SVE scalable vector register is 2048 bits.

R_{RXPHX} The minimum size of an SVE scalable vector register is 128 bits.

I_{GKWYJ} Unless stated otherwise in an instruction description, SVE instructions treat an SVE scalable vector register as containing one or more vector elements that are equal in size.

I_{CDKJQ} Unless stated otherwise in an instruction description, vector elements can be processed in parallel by SVE instructions.

R_{KHDBN} When an SVE scalable vector register is divided into vector elements by an instruction, the size of the vector elements is encoded in the opcode of the instruction. The size of the vector elements is 8, 16, 32, 64, or 128 bits.

R_{CJZLM} When the order of operations performed by an SVE instruction on vector or predicate elements has observable significance, elements are processed in increasing element number order.

I_{DBZRX} The layouts of an SVE 256-bit vector register and a SIMD&FP vector in AArch64 state are:

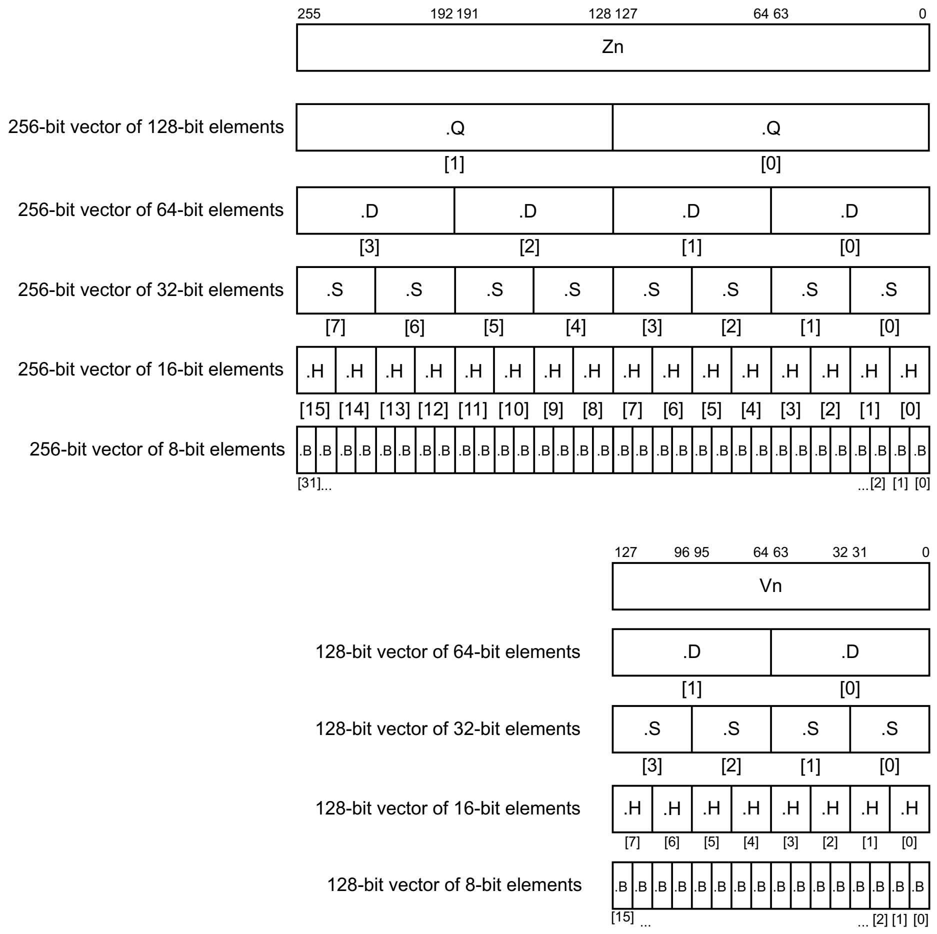

255
192 191
128 127
64 63
0
Zn
256-bit vector of 128-bit elements
.Q
[1] [0]
256-bit vector of 64-bit elements
.D
[3] [2] [1] [0]
256-bit vector of 32-bit elements
.S
[7] [6] [5] [4] [3] [2] [1] [0]
256-bit vector of 16-bit elements
.H
[15] [14] [13] [12] [11] [10] [9] [8] [7] [6] [5] [4] [3] [2] [1] [0]
256-bit vector of 8-bit elements
.B
[31]... ...[2] [1] [0]
127
96 95
64 63
32 31
0
Vn
128-bit vector of 64-bit elements
128-bit vector of 32-bit elements
128-bit vector of 16-bit elements
128-bit vector of 8-bit elements
[15] ... ... [2] [1] [0]

Figure B1-4 SVE vectors in AArch64 state

R_{YDXCP} Bits[127:0] of each of the SVE scalable vector registers, Z0-Z31, hold the correspondingly numbered AArch64 SIMD&FP register, V0-V31.

R_{WKYLB} When the accessible SVE vector length at the current Exception level is greater than 128 bits, any AArch64 instruction that writes to V0-V31 sets all the accessible bits above bit [127] of the corresponding SVE scalable vector register to zero.

SVE writes to scalar registers

I_{ZDLGD} Certain SVE instructions generate a scalar result that is written to an AArch64 general-purpose register or to element[0] of a vector register.

R_{HNVTM} When an SVE instruction generates a scalar result of width N bits, the instruction places the result in bits [N-1:0] of the destination register.

R_{QCLSH} When an instruction generates a scalar result of width N bits, and N is less than the maximum accessible destination register width RW, the instruction sets bits [RW-1:N] of the destination register to zero.

B1.2.3 SVE predicate registers

R_{DCWFB} SVE has 16 scalable predicate registers named P0-P15.

R_{NLGZS} Each SVE predicate register holds one bit for each byte of a vector register.

R_{NKRJV} The size of an SVE predicate register is an IMPLEMENTATION DEFINED multiple of 16 bits.

R_{MFPXG} The maximum size of an SVE predicate register is 256 bits.

R_{BBTXX} The minimum size of an SVE predicate register is 16 bits.

R_{XVRKX} Unless stated otherwise in the instruction description, SVE instructions treat an SVE predicate register as containing one or more predicate elements of equal size.

R_{XMPLM} Each predicate register can be subdivided into a number of 1-bit, 2-bit, 4-bit, or 8-bit elements.

R_{NSXCV} Each predicate element in a predicate register corresponds to a vector element.

R_{XCZQR} When a predicate register is divided into predicate elements by an instruction, the size of the predicate elements is encoded in the opcode of the instruction.

R_{DNMFH} If the lowest-numbered bit of a predicate element is 1, the value of the predicate element is TRUE.

R_{HRPMD} If the lowest-numbered bit of a predicate element is 0, the value of the predicate element is FALSE.

R_{HBMLS} For all SVE instructions, if all of the following are true, all bits except the lowest-numbered bit of each predicate element are ignored on reads:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

R_{LTGQC} For all SVE instructions, if all of the following are true, all bits except the lowest-numbered bit of each predicate element are set to zero on writes:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

B1.2.4 FFR, First Fault Register

R_{TRLWH} SVE has a dedicated First Fault Register named FFR.

I_{XPLQW} The FFR captures the cumulative fault status of a sequence of SVE First-fault and Non-fault vector load instructions.

R_{CPQQN} The FFR and the predicate registers have the same size and format.

I_{PBWPM} The FFR is a Special-purpose register.

R_{CGHCK} All bits in the FFR that are accessible at the current Exception level are initialized to 1 by using the `SETFFR` instruction.

R_{WZJVT} Bits in the FFR are indirectly set to 0 as a result of a suppressed access or fault generated in response to an *Active element* of an SVE First-fault or Non-fault vector load.

R_{BZLJG} Bits in the FFR are never set to 1 as a result of a vector load instruction.

I_{XLZQY} After a sequence of one or more SVE First-fault or Non-fault loads that follow a `SETFFR` instruction, the FFR contains a sequence of zero or more TRUE elements, followed by zero or more FALSE elements.

I_{TQMTV} The TRUE elements in the FFR indicate the shortest sequence of consecutive elements that could contain valid data loaded from memory.

R_{GHFRQ} The only instructions that directly read the FFR are:

- `RDFFR (predicated)`.
- `RDFFRS`.

R_{LHBRN} The only instructions that directly write the FFR are:

- `WRFFR`.
- `SETFFR`.

R_{XXMMP} All direct and indirect reads and writes to the FFR occur in program order relative to other instructions, without explicit synchronization.

B1.2.5 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Condition flags and related instructions* on page C6-1141.

The exception masking bits

D Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

A SError interrupt mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0	Exception not masked.
1	Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

See *Process state, PSTATE* on page D1-4633 for the system level view of PSTATE.

Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using AArch64. All other PSTATE fields do not have direct read and write access at EL0.

Table B1-1 Accessing PSTATE fields at EL0 using MRS and MSR (register)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F}. Table B1-2 shows the MSR (immediate) operands that can directly write to PSTATE.{D, A, I, F} when the PE is at EL0 using AArch64 state.

Table B1-2 Accessing PSTATE.{D, A, I, F} at EL0 using MSR (immediate)

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Directly sets any of the PSTATE.{D,A, I, F} bits to 1
DAIFClr	D, A, I, F	Directly clears any of the PSTATE.{D, A, I, F} bits to 0

However, access to the PSTATE.{D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

SVE use of PSTATE N, Z, C, and V Condition flags

IWYXLS This section describes the SVE-specific use of PSTATE.

IYZYCQ PSTATE N, Z, C and V condition flags can be updated by any of the following:

- An SVE instruction that generates a predicate result and updates the PSTATE N, Z, C and V Condition flags based on the value of the result.
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the value in its predicate source register or FFR:
  - `PTEST`.
  - `RDFFR (predicated)`.
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the values in its general-purpose source registers:
  - `CTERMEQ, CTERMNE`.

RTPXTF When setting the PSTATE N, Z, C and V Condition flags for SVE predicated flag-setting instructions, the instruction's *Governing predicate* determines which predicate elements are considered Active.

RQJBRW When setting the PSTATE N, Z, C and V Condition flags for SVE unpredicated flag-setting instructions, all predicate elements are considered Active.

RZMRXC Unless otherwise specified in an instruction description, the SVE flag-setting instructions update the PSTATE N, Z, C and V Condition flags as follows:

Flag	SVE Name	SVE interpretation
N	First	Set to 1 if the *First active element* was TRUE, otherwise cleared to 0.
Z	None	Cleared to 0 if any *Active element* was TRUE, otherwise set to 1.
C	Not last	Cleared to 0 if the *Last active element* was TRUE, otherwise set to 1.
V	-	Cleared to 0.

B1.2.6 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register with the access right disabled causes the instruction to behave as UNDEFINED. The registers that can be accessed from EL0 are:

Cache ID registers The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.

Debug registers A Debug Communications Channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0 and DBGDTRTX_EL0 registers.

Performance Monitors registers

The Performance Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure some of these registers to be accessible at EL0.

For more details, see Chapter D8 *The Performance Monitors Extension*.

Activity Monitors registers

The Activity Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure these registers to be accessible at EL0.

For more details, see Chapter D9 *The Activity Monitors Extension*.

Thread ID registers The TPIDR_EL0 and TPIDRRO_EL0 registers are two thread ID registers with different access rights.

Timer registers The following operations are performed by these registers:

- Read access to the system counter clock frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0 and CNTVCT_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

B1.3 Software control features and EL0

The following sections describe the EL0 view of the software control features:

- *Exception handling*
- *Wait for Interrupt and Wait for Event*
- *The YIELD instruction*
- *Application level cache management* on page B1-145
- *Instructions relating to Debug* on page B1-145
- *About PSTATE.DIT* on page B1-123

B1.3.1 Exception handling

In the Arm architecture, an *exception* causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:

- Interrupts.
- Memory system aborts.
- Exceptions generated by attempting to execute an instruction that is UNDEFINED.
- System calls.
- Secure monitor or Hypervisor traps.
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in Chapter D1 *The AArch64 System Level Programmers' Model*.

The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

The `BRK` instruction generates a Breakpoint Instruction exception. This provides a mechanism for debugging software using debugger executing on the same PE, see *Breakpoint Instruction exceptions* on page D2-4669.

Note

The `BRK` instruction is supported only in the A64 instruction set. The equivalent instruction in the T32 and A32 instruction sets is `BKPT`.

B1.3.2 Wait for Interrupt and Wait for Event

Issuing a `WFI` instruction indicates that no further execution is required until a WFI wake-up event occurs, see *Wait for Interrupt mechanism* on page D1-4641. This permits entry to a low-power state.

Issuing a `WFE` instruction indicates that no further execution is required until a WFE wake-up event occurs, see *Wait for Event* on page D1-4639. This permits entry to a low-power state.

B1.3.3 The YIELD instruction

The `YIELD` instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see *YIELD* on page C6-2011. This mechanism can be used to improve overall performance in a *Symmetric Multithreading* (SMT) or *Symmetric Multiprocessing* (SMP) system.

Examples of when the `YIELD` instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The `YIELD` instruction permits binary compatibility between SMT and SMP systems.

The `YIELD` instruction is a `NOP` hint instruction.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

B1.3.4 Application level cache management

A small number of cache management instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation with the access right disabled causes the instruction to behave as UNDEFINED.

About the available operations, see *Application level access to functionality related to caches* on page B2-182.

B1.3.5 Instructions relating to Debug

Exception handling on page B1-144 refers to the `BRK` instruction, which generates a Breakpoint Instruction exception. In addition, in both AArch64 state and AArch32 state, the `HLT` instruction causes the PE to halt execution and enter Debug state. This provides a mechanism for debugging software using a debugger that is external to the PE, see Chapter H1 *About External Debug*.

Note

In AArch32 state, previous versions of the architecture defined the `DBG` instruction, which could provide a hint to the debug system. This instruction executes as a `NOP`. Arm deprecates the use of the `DBG` instruction.

B1.3.6 About PSTATE.DIT

When the value of PSTATE.DIT is 1:

- The instructions listed in DIT are required to have;
  - Timing which is independent of the values of the data supplied in any of its registers, and the values of the NZCV flags.
  - Responses to asynchronous exceptions which do not vary based on the values supplied in any of their registers, or the values of the NZCV flags.
- All loads and stores must have their timing insensitive to the value of the data being loaded or stored.
- If SVE2 is implemented, the data independent timing of the following subset of SVE and SVE2 instructions is affected in all of the following ways:
  - For unpredicated SVE and SVE2 instructions, the instruction timing is independent of all of the following:
    - The data values supplied in any of its operand registers.
    - The values of the NZCV flags.
  - For predicated SVE and SVE2 instructions, the instruction timing is independent of all of the following:
    - The data values supplied in any of its operand registers when its *Governing predicate* register contains the same value for each execution.
    - The values of the NZCV flags.
  - For predicated SVE and SVE2 instructions, the architecture does not require that the instruction timing is independent of the value of the *Governing predicate*.
  - For the SVE `SEL` instruction, the instruction timing is independent of all of the following:
    - The data values supplied in any of its operand registers.
    - The values of the NZCV flags.

Note

- The use of value prediction for load data values when PSTATE.DIT is set, is not compatible with the requirement that the timing is insensitive to the data value being loaded.
- Arm recommends that the FEAT_PAuth instructions do not have their timing dependent on the key value used in the pointer authentication, regardless of the PSTATE.DIT bit.
- When the value of PSTATE.DIT is 0, the architecture makes no statement about the timing properties of any instructions. However, it is likely that these instructions have timing that is invariant of the data in many situations.
- If SVE2 is not implemented, the data independent timing control introduced by FEAT_DIT does not affect the timing properties of *SVE* instructions.
- The *Operational information* section of an SVE or an SVE2 instruction description indicates whether or not that instruction honors the PSTATE.DIT control. If the *Operational information* section of an SVE instruction description does not mention PSTATE.DIT or if the section does not exist, then the instruction timing is not affected by PSTATE.DIT.
- For SVE and SVE2 predicated instructions, it is the programmer's responsibility to use a *Governing predicate* that does not reflect the values of the data being operated on.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

On an exception that is taken from AArch64 state to AArch64 state, PSTATE.DIT is copied to SPSR_ELx.DIT.

On an exception that is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

On an exception return from AArch64 state:
- SPSR_ELx.DIT is copied to PSTATE.DIT, when the target Exception level is in AArch64 state.
- SPSR_ELx.DIT is copied to CPSR.DIT, when the target Exception level is in AArch32 state.

PSTATE.DIT can be written and read at all Exception levels.

Note
- PSTATE.DIT is unchanged on entry into Debug state.
- PSTATE.DIT is not guaranteed to have any effect in Debug state.

B1.4 SVE predicated instructions

I_{VKHDR} If an instruction supports predication, it is known as a predicated instruction.

I_{FNFRN} The predicate operand that is used to determine the *Active elements* of a predicated instruction is known as the *Governing predicate*.

I_{SVYXB} An instruction that does not have a *Governing predicate* operand and implicitly treats all other vector and predicate elements as Active is known as an unpredicated instruction.

I_{KNKBN} Many predicated instructions can only use P0-P7 as the *Governing predicate*.

R_{LZVFJ} When a *Governing predicate* element is TRUE, the corresponding element in other vector or predicate operands is an *Active element*.

R_{CNFLG} When a *Governing predicate* element is FALSE, the corresponding element in other vector or predicate operands is an *Inactive element*.

R_{CBYJH} Predicated instructions process *Active elements*.

R_{LDXSF} Predicated instructions do not process *Inactive elements*.

R_{GJLPZ} Unpredicated instructions process all elements in their vector or predicate operands.

R_{WLQBD} When a predicated instruction writes to a vector destination register or a predicate destination register, one of the following happens:

- The *Inactive elements* in the destination register are set to zero.
- The *Inactive elements* in the destination register retain their previous value.

I_{QBHRN} Zeroing predication is performed when the *Inactive elements* in the destination register are set to zero.

I_{YPYRF} Merging predication is performed when *Inactive elements* in the destination register retain their previous value.

Chapter B2
The AArch64 Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:

Note

In this chapter, System register names usually link to the description of the register in Chapter D13 *AArch64 System Register Descriptions*, for example. SCTLR_EL1.

B2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be reordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.
- If FEAT_MTE2 is implemented, the definitions of the memory model which apply to data accesses and data apply to Allocation Tag accesses and Allocation tags.

This section contains:

- *Address space*.
- *Memory type overview*.

B2.1.1 Address space

Address calculations are performed using 64-bit registers. However, supervisory software can configure the top eight address bits for use as a tag, as described in *Address tagging in AArch64 state* on page D5-4770. If this is done, address bits[63:56]:

- Are not considered when determining whether the address is valid.
- Are never propagated to the program counter.

Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

Simple sequential execution of instructions might overflow the valid address range. For more information, see *Virtual address space overflow* on page D4-4729.

Memory accesses use the `Mem[]` function. This function makes an access of the required type. If supervisory software configures the top eight address bits for use as a tag, the top eight address bits are ignored.

The `AccType{}` enumeration defines the different access types.

Note

- Chapter D4 *The AArch64 System Level Memory Model* and Chapter D5 *The AArch64 Virtual Memory System Architecture* include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also include pseudocode descriptions of these operations.
- For information on the pseudocode that relates to memory accesses, see *Basic memory access* on page D4-4763, *Unaligned memory access* on page D4-4763, and *Aligned memory access* on page D4-4763.

B2.1.2 Memory type overview

The Arm architecture provides the following mutually-exclusive memory types:

Normal This is generally used for bulk memory operations, both read/write and read-only operations.

Device The Arm architecture forbids *Speculative* reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive Locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See *Gathering* on page B2-200.
- They preserve the access order and synchronization requirements for accesses to a single peripheral. See *Reordering* on page B2-201.
- They indicate whether a write can be acknowledged other than at the end point. See *Early Write Acknowledgement* on page B2-202.

For more information on Normal memory and Device memory, see *Memory types and attributes* on page B2-193.

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A *Note* in *Device memory* on page B2-197 describes how these memory types map onto the Armv8 memory types.

B2.1.3 SVE memory model

ICZFSY SVE predicated memory operations have a vector element size and a memory element access size. The vector element size specifies the data that is read from and written to the vector. The memory element access size specifies the amount of data that is read from and written to the memory.

ITJQJF The vector element size and the memory element access size do not need to have the same value.

ILGGHH For each memory element, there is an associated element address.

SVE also affects behavior in the following areas:

- *Requirements for single-copy atomicity* on page B2-152.
- *SVE memory ordering relaxations* on page B2-168.
- *Load or Store of Single or Multiple registers* on page B2-186.
- *Endianness in SVE operations* on page B2-191.
- *SVE loads and stores that access Device memory* on page B2-203.

B2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. In the Arm architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- *Requirements for single-copy atomicity*.
- *Properties of single-copy atomic accesses* on page B2-154.
- *Multi-copy atomicity* on page B2-154.
- *Requirements for multi-copy atomicity* on page B2-154.
- *Concurrent modification and execution of instructions* on page B2-154.

For more information about the memory types, see *Memory type overview* on page B2-150.

B2.2.1 Requirements for single-copy atomicity

For explicit memory effects generated from an Exception level the following rules apply:

- A read that is generated by a load instruction that loads a single general-purpose register and is aligned to the size of the read in the instruction is single-copy atomic.
- A write that is generated by a store instruction that stores a single general-purpose register and is aligned to the size of the write in the instruction is single-copy atomic.
- Reads that are generated by a Load Pair instruction that loads two general-purpose registers and are aligned to the size of the load to each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes that are generated by a Store pair instruction that stores two general-purpose registers and are aligned to the size of the store of each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-Exclusive Pair instructions of two 32-bit quantities and Store-Exclusive Pair instructions of 32-bit quantities are single-copy atomic.
- When the Store-Exclusive of a Load-Exclusive/Store-Exclusive pair instruction using two 64-bit quantities succeeds, it causes a single-copy atomic update of the entire memory location being updated.

  Note

  To atomically load two 64-bit quantities, perform a Load-Exclusive pair/Store-Exclusive pair sequence of reading and writing the same value for which the Store-Exclusive pair succeeds, and use the read values from the Load-Exclusive pair.

- Where translation table walks generate a read of a translation table entry, this read is single-copy atomic.
- For the atomicity of instruction fetches, see *Concurrent modification and execution of instructions* on page B2-154.
- Reads to SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being stored, have each element treated as a single-copy atomic store.
- Reads to SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit reads.
- Writes from SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit writes.
- When FEAT_SVE is implemented, atomicity rules for SIMD load and store instructions also apply to SVE load and store instructions.

- When FEAT_SVE is implemented, SVE predicated load and store instructions are performed as a sequence of memory element accesses.
- When FEAT_SVE is implemented, if an SVE predicated load or store instruction uses an element address that is aligned to the specified memory element access size, the related element memory access is performed as a single-copy atomic access.
- When FEAT_SVE is implemented, SVE unpredicated load and store instructions are performed as a sequence of byte accesses.
- When FEAT_SVE is implemented, SVE unpredicated load and store instructions do not guarantee that any access larger than a byte will be performed as a single-copy atomic access.
- When FEAT_LS64 is implemented, a single-copy atomic load of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte read from the target address.
- When FEAT_LS64 is implemented, a single-copy atomic store of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte write to the target address.
- For unaligned memory accesses, the single-copy atomicity is described in *Alignment of data accesses* on page B2-186.
- The reads and writes of the two words or two double-words accessed by CASP instructions are single-copy atomic at the size of the two words or double-words.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

Note

In AArch64 state, no memory accesses from a DC ZVA have single-copy atomicity of any quantity greater than individual bytes.

If, according to these rules, an instruction is executed as a sequence of accesses, exceptions, including interrupts, can be taken during that sequence, regardless of the memory type being accessed. If any of these exceptions are returned from using their preferred return address, the instruction that generated the sequence of accesses is re-executed, and so any access performed before the exception was taken is repeated. See also *Taking an interrupt during a multi-access load or store* on page D1-4627.

Note

The exception behavior for these multiple access instructions means that they are not suitable for use for writes to memory for the purpose of software synchronization.

Changes to single-copy atomicity in Armv8.4

In addition to the single-copy atomicity requirements listed above:

Instructions that are introduced in FEAT_LRCPC are single-copy atomic when all of the following conditions are true:

- All bytes being accessed are within the same 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, all loads and stores are single-copy atomic when all of the following conditions are true:

- Accesses are unaligned to their data size but all bytes being accessed are within a 16-byte quantity that is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, `LDP`, `LDNP`, and `STP` instructions that load or store two 64-bit registers are single-copy atomic when all of the following conditions are true:

- The overall memory access is aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, `LDP`, `LDNP`, and `STP` instructions that access fewer than 16 bytes are single-copy atomic when all of the following conditions are true:

- All bytes being accessed are within a 16-byte quantity aligned to 16 bytes.
- Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

B2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2, if one of the overlapping reads generated by L_1 Reads-from one of the overlapping writes generated by S_2, then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1.

For more information, see *Definition of the Arm memory model* on page B2-157.

B2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

B2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The Arm memory model is Other-multi-copy atomic. For more information, see *External ordering constraints* on page B2-164.

B2.2.5 Concurrent modification and execution of instructions

The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where each of the instruction before modification and the instruction after modification is one of a `B`, `B.cond`, `BL`, `BRK`, `CBNZ`, `CBZ`, `HVC`, `ISB`, `NOP`, `SMC`, `SVC`, `TBNZ` or `TBZ` instruction.

For the `B`, `B.cond`, `BL`, `BRK`, `CBNZ`, `CBZ`, `HVC`, `ISB`, `NOP`, `SMC`, `SVC`, `TBNZ` and `TBZ` instructions, the architecture guarantees that after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.

2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH
```

Note

- The `DC CVAU` operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.

3. In a multiprocessor system, the `IC IVAU` is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, when the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
ISB                                ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see *Synchronization and coherency issues between data and instruction accesses* on page B2-184.

For information about memory accesses caused by instruction fetches, see *Ordering relations* on page B2-161.

B2.2.6 Possible implementation restrictions on using atomic instructions

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

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

In such implementations, it is defined by the system:

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

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

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

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

The architecture only requires that *Conventional memory* that is mapped in this way supports this functionality.

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

- The instruction generates a synchronous External abort.
- The instruction generates a System Error interrupt.

- The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

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

B2.3 Definition of the Arm memory model

This section describes observation and ordering in the Arm memory model. It contains the following subsections:

- *Basic definitions*.
- *Dependency definitions* on page B2-160.
- *Ordering relations* on page B2-161.
- *Ordering constraints* on page B2-163.
- *Internal visibility requirement* on page B2-164.
- *External ordering constraints* on page B2-164.
- *Completion and endpoint ordering* on page B2-166.
- *Ordering of instruction fetches* on page B2-168.
- *Restrictions on the effects of speculation* on page B2-169.
- *Memory barriers* on page B2-172.
- *Limited ordering regions* on page B2-179.

For more information about endpoint ordering of memory accesses, see *Reordering* on page B2-201.

In the Arm memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see *Memory types and attributes* on page B2-193.

The Arm architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- Chapter D4 *The AArch64 System Level Memory Model*.
- Chapter D5 *The AArch64 Virtual Memory System Architecture*.

See also *Mismatched memory attributes* on page B2-205.

B2.3.1 Basic definitions

The Arm memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory.

Observer

An *Observer* refers to a processing element or mechanism in the system, such as a peripheral device, that can generate reads from, or writes to, memory.

Common Shareability Domain

For the purpose of this section, all Observers are assumed to belong to a Common Shareability Domain. All read and write effects access only Normal memory locations in a Common Shareability Domain, and excludes the situations described in *Mismatched memory attributes* on page B2-205.

Location

A *Location* is a byte that is associated with an address in the physical address space.

Note

It is expected that an operating system will present the illusion to the application programmer that is consistent with a location also being considered as a byte that is associated with an address in the virtual address space.

Effects

The *Effects* of an instruction can be:

- Register effects.
- Memory effects.
- Barrier effects.
- Tag effects.
- Branching effects.

The effects of an instruction I_1 are said to appear in program order before the effects of an instruction I_2 if and only if I_1 occurs before I_2 in the order specified by the program. Each effect generated by an instruction has a unique identifier, which characterizes it amongst the events generated by the same instruction.

Register effect

The *Register effects* of an instruction are register reads or register writes of that instruction. For an instruction that accesses registers, a register read effect is generated for each register read by the instruction and a register write effect is generated for each register written by the instruction. An instruction may generate both read and write Register effects.

Memory effect

The *Memory effects* of an instruction are the memory reads or writes generated by that instruction. For an instruction that accesses memory, a memory read effect is generated for each Location read by the instruction and a memory write effect is generated for each Location written by the instruction. An instruction may generate both read and write Memory effects.

Tag effect

The *Tag effects* of a Memory Tagging instruction are the memory read or write effects of that instruction that affect tag locations.

Tag-read

A *Tag-read* is a read of a tag location generated by an `LDG` instruction.

Tag-write

A *Tag-write* is a write of a tag location generated by an `STG` instruction.

Tag-Check-read

A *Tag-Check-read* is a read of a tag location that is generated by a checked memory access. All other reads and writes are considered Data accesses.

Branching effect

The *Branching effects* of an instruction are effects which correspond to a branching decision being taken.

Note

Conditional and compare-and-swap instructions do not create Branching effects.

Intrinsic order

There is a per-instruction *Intrinsic order* relation that provides a partial order over the effects of that instruction, according to the operation of that instruction.

The operation of an instruction is defined by the pseudocode in Chapter C6 *A64 Base Instruction Descriptions*.

Reads-from-register

The *Reads-from-register* relation couples register read and write effects to the same register such that each register read effect is paired with exactly one register write effect in the execution of a program. A register read effect R_2 Reads-from-register a register write effect W_1 to the same register if and only if R_2 takes its data from W_1. By construction W_1 must be in program order before R_2 and there must be no intervening write to the same register in program order between W_1 and R_2.

Reads-from

The *Reads-from* relation couples memory read and write effects to the same Location such that each memory read effect is paired with exactly one memory write effect in the execution of a program. A memory read effect R_2 from a Location Reads-from a memory write effect W_1 to the same Location if and only if R_2 takes its data from W_1.

Coherence order

There is a per-location *Coherence order* relation that provides a total order over all memory write effects from all coherent Observers to that Location, starting with a notional memory write effect of the initial value. The Coherence order of a Location represents the order in which memory write effects to the Location arrive at memory.

Local read successor

A memory read effect R_2 of a Location is the *Local read successor* of a memory write effect W_1 from the same Observer to the same Location if and only if W_1 appears in program order before R_2 and there is not a memory write effect W_3 from the same Observer to the same Location appearing in program order between W_1 and R_2.

Local write successor

A memory write effect W_2 of a Location is a *Local write successor* of a memory read or write effect RW_1 from the same Observer to the same Location if and only if RW_1 appears in program order before W_2.

Coherence-after

A memory write effect W_2 to a Location is *Coherence-after* another memory write effect W_1 to the same Location if and only if W_2 is sequenced after W_1 in the Coherence order of the Location.

A memory write effect W_2 to a Location is Coherence-after a memory read effect R_1 of the same location if and only if R_1 Reads-from a memory write effect W_3 to the same Location and W_2 is Coherence-after W_3.

Observed-by

A memory read or write effect RW_1 from an Observer is *Observed-by* a memory write effect W_2 from a different Observer if and only if W_2 is coherence-after RW_1.

A memory write effect W_1 from an Observer is Observed-by a memory read effect R_2 from a different Observer if and only if R_2 Reads-from W_1.

Note

The Observed-by relation relates only Memory effects generated by different Observers.

Overlapping accesses

Two Memory effects overlap if and only if they access the same Location. Two instructions overlap if and only if one or more of their generated Memory effects overlap.

Single-copy-atomic-ordered-before

A memory read effect R_1 is *Single-copy-atomic-ordered-before* another memory read effect R_2 if and only if all of the following statements are true:

- R_1 and R_2 are memory read effects generated by the same instruction.
- R_1 is not a Local read successor of a memory write effect.
- R_2 is a Local read successor of a memory write effect.

`DMB FULL`

A `DMB FULL` is a `DMB` with neither the `LD` or the `ST` qualifier.

Where this section refers to `DMB` without any qualification, then it is referring to all types of `DMB`. Unless a specific shareability domain is defined, a `DMB` applies to the Common Shareability Domain.

All properties that apply to `DMB` also apply to the corresponding `DSB`.

Context synchronization instruction

A *Context synchronization instruction* is one of the following:

- An `ISB` instruction.
- An instruction that generates a synchronous exception.

- An exception return instruction.
- A `DCPS` or `DRPS` instruction.

B2.3.2 Dependency definitions

Dependency through registers

A *Dependency through registers* from a first effect E_1 to a second effect E_2 exists within a PE if and only if at least one of the following applies:

- E_1 is a register write effect W_1 which has not been generated by a Store Exclusive, E_2 is a register read effect R_2 and R_2 Reads-from-register W_1.
- E_1 and E_2 have been generated by the same instruction and E_1 is before E_2 in the Intrinsic order of that instruction.
- There is a Dependency through registers from E_1 to a third effect E_3, and there is a Dependency through registers from E_3 to E_2.

Address dependency

An *Address dependency* from a memory read effect R_1 to a Memory effect RW_2 exists if and only if there is a Dependency through registers from R_1 to a Register effect E_3 generated by RW_2, and E_3 affects the address part of RW_2, and either:

- RW_2 is a memory write effect W_2.
- RW_2 is a memory read effect R_2 and there is no Branching effect D_4 such that there is a Dependency through registers from R_1 to D_4 and from D_4 to R_2.

Note

An Address dependency exists from a memory read effect R_1 to a Tag-Check-read R_2 if and only if there is a Dependency through registers from R_1 to the address part of R_2.

Data dependency

A *Data dependency* from a memory read effect R_1 to a memory write effect W_2 exists if and only if there is a Dependency through registers from R_1 to a Register effect E_3 generated by W_2, and E_3 affects the data part of W_2.

Control dependency

A *Control dependency* from a memory read effect R_1 to a Memory effect RW_2 exists if and only if either:

- There is a Dependency through registers from R_1 to a Branching effect B_3 and B_3 is in program order before RW_2.
- There is a Dependency through registers from R_1 to the determination of a synchronous exception on an instruction generating an effect RW_3, and RW_2 appears in program order after RW_3.

Note

This notion is under review. Arm’s intent is that a branch instruction between a read and a write, where the branch condition is dependent on the read, will provide order, regardless of whether the branch is taken. This applies only to branch instructions and not to conditional selection or other conditional data processing instructions. A formal definition of this change will be issued soon as an erratum to the Arm Architecture Reference Manual for A-profile architecture.

Pick Basic dependency

A *Pick Basic dependency* from a read Register effect or read Memory effect R_1 to a Register effect or Memory effect E_2 exists if and only if all of the following apply:

- There is a Dependency through registers from R_1 to a Register effect E_3, and one of the following applies:
  - There is an Intrinsic control dependency from the Register effect E_3 to a Register effect E_4, and there is a chain of Dependency through registers or Dependency through memory from E_4 to E_2.
  - There is an Intrinsic control dependency from the Register effect E_3 to E_2.

Pick Address dependency

A *Pick Address dependency* from a read Register effect or read Memory effect R_1 to a read or write Memory effect RW_2 exists if and only if all of the following apply:

- There is a Pick Basic dependency from R_1 to a Register effect E_3.
- There is an Intrinsic data dependency from the Register effect E_3 to RW_2.
- The Register effect E_3 affects the address of the location accessed by RW_2.

Pick Data dependency

A *Pick Data dependency* from a read Register effect or read Memory effect R_1 to a write Memory effect W_2 exists if and only if all of the following apply:

- There is a Pick Basic dependency from R_1 to a Register effect E_3.
- There is an Intrinsic data dependency from the Register effect E_3 to RW_2.
- The Register effect E_3 affects the value written by W_2.

Pick Control dependency

A *Pick Control dependency* from a read Register effect or read Memory effect R_1 to an effect E_2 exists if and only if all of the following apply:

- There is a Pick Basic dependency from R_1 to a Branching effect BR_3.
- The Branching effect BR_3 is in program order before E_2.

Pick dependency

A *Pick dependency* from a read Register effect or read Memory effect R_1 to an effect E_2 exists if and only if one of the following applies:

- There is a Pick Basic dependency from R_1 to E_2.
- There is a Pick Address dependency from R_1 to E_2.
- There is a Pick Data dependency from R_1 to E_2.
- There is a Pick Control dependency from R_1 to E_2.

B2.3.3 Ordering relations

Dependency-ordered-before

A dependency creates externally-visible order between a memory read effect and another Memory effect generated by the same Observer. A memory read effect R_1 is *Dependency-ordered-before* a memory read or write effect RW_2 from the same Observer if and only if R_1 appears in program order before RW_2 and any of the following cases apply:

- There is an Address dependency or a Data dependency from R_1 to RW_2.
- RW_2 is a memory write effect W_2 and there is a Control dependency from R_1 to W_2.
- RW_2 is a memory read effect R_2 generated by an instruction appearing in program order after an instruction that generates a Context synchronization event E_3, and there is a Dependency through registers from R_1 to E_3.

- RW_2 is a memory write effect W_2 appearing in program order after a memory read or write effect RW_3 and there is an Address dependency from R_1 to RW_3.
- RW_2 is a Local read successor R_2 of a memory write effect W_3 and there is an Address dependency or a Data dependency from R_1 to W_3.

Pick-ordered-before

A read Register effect or read Memory effect R_1 is *Pick-ordered-before* a read or write Memory effect RW_2 from the same Observer if and only if R_1 appears in program order before RW_2 and any of the following cases apply:

- RW_2 is a write Memory effect W_2 and there is a Pick dependency from R_1 to W_2.
- RW_2 is a read Memory effect R_2 generated by an instruction appearing in program order after an instruction that generates a Context synchronization event E_3, and there is a Pick Control dependency from R_1 to E_3.
- RW_2 is a Memory effect generated by an instruction appearing in program order after an instruction that generates a Context synchronization event E_3, there is a Pick Address dependency from R_1 to an effect E_4 and E_4 is in program order before E_3.
- RW_2 is a write Memory effect W_2, there is a Pick Address dependency from R_1 to a read or write Memory effect RW_3 and W_2 is program-order-after RW_3.

Atomic-ordered-before

Load-Exclusive and Store-Exclusive instructions provide some ordering guarantees, even in the absence of dependencies. A read or a write Memory effect RW_1 is *Atomic-ordered-before* a read or a write Memory effect RW_2 from the same Observer if and only if RW_1 appears in program order before RW_2 and either of the following cases apply:

- RW_1 is a read Memory effect R_1 and RW_2 is a write Memory effect W_2 such that R_1 and W_2 are generated by an atomic instruction or a successful Load-Exclusive/Store-Exclusive instruction pair to the same Location.
- RW_1 is a read Memory effect R_1 generated by an atomic instruction (resp. a Load-Exclusive instruction) and RW_2 is a read Memory effect R_2 generated by an instruction with Acquire or AcquirePC semantics such that R_2 is a Local read successor of the write Memory effect W_3 generated by the same atomic instruction as R_1 (resp. the successful Store-Exclusive instruction paired with the Load-Exclusive instruction that generated R_1).

For more information, see *Synchronization and semaphores* on page B2-208.

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A memory read or write effect RW1 is *Barrier-ordered-before* a memory read or write effect RW_2 from the same Observer if and only if RW1 appears in program order before RW_2 and any of the following cases apply:

- RW_1 appears in program order before a `DMB FULL` that appears in program order before RW_2.
- RW_1 is a memory write effect W_1 and is generated by an atomic instruction with both Acquire and Release semantics.
- RW_1 is a memory write effect W_1 generated by an instruction with Release semantics and RW_2 is a memory read effect R_2, except a Tag-Check-read, generated by an instruction with Acquire semantics.
- RW_1 is a memory read effect R_1 and appears in program order before a `DMB LD` that appears in program order before RW_2.
- RW_1 is a memory read effect R_1, except a Tag-Check-read, and is generated by an instruction with Acquire or AcquirePC semantics.
- RW_1 is a memory write effect W_1 and RW_2 is a memory write effect W_2 appearing in program order before a `DMB ST` that appears in program order before W_2.
- RW_2 is a memory write effect W_2 and is generated by an instruction with Release semantics.

Tag-ordered-before

If FEAT_MTE2 is implemented, a Tag read R_1 is *Tag-ordered-before* a memory read or write effect Checked data access RW_2 generated by the same instruction if and only if all of the following apply:

- R_1 is in the Intrinsic order of that instruction before RW_2.
- R_1 reads the Allocation Tag at a tag physical address and compares it with the physical address Tag of the instruction. If the result of the comparison can cause a precise exception and the result is negative, then RW_2 does not architecturally occur.

Tag-Location-Ordered

Tag-Check-reads R_1 and R_2 are *Tag-Location-Ordered* if and only if all the following apply:

- R_1 is Tag-ordered-before a Checked data access RW_3.
- R_2 is Tag-ordered-before a Checked data access RW_4.
- RW_3 and RW_4 are to the same Location.

Locally-ordered-before

Dependencies, Local write successor, load/store-exclusive, atomic and barrier instructions can be composed within an Observer to create externally-visible order. A memory read or write effect RW_1 is *Locally-ordered-before* a memory read or write effect RW_2 from the same Observer if and only if any of the following cases apply:

- RW_1 is a memory write effect W_1 and RW_2 is a memory write effect W_2 that is equal to or generated by the same instruction as a Local write successor of RW_1.
- RW_1 is Pick-ordered-before RW_2.
- RW_1 is Dependency-ordered-before RW_2.
- RW_1 is Atomic-ordered-before RW_2.
- RW_1 is Barrier-ordered-before RW_2.
- RW_1 is Tag-ordered-before RW_2.
- RW_1 is Locally-ordered-before a memory read or write effect that is Locally-ordered-before RW_2.

Pick-locally-ordered-before

A read or a write Memory effect RW_1 is *Pick-locally-ordered-before* a read or a write Memory effect RW_2 from the same Observer if and only if all of the following apply:

- There is a Pick Basic dependency from RW_1 to an effect E_3.
- RW_2 is a write Memory effect W_2.
- E_3 is Locally-ordered-before W_2.

B2.3.4 Ordering constraints

The Arm memory model is described as being Other-multi-copy atomic. The definition of Other-multi-copy atomic is as follows:

Other-multi-copy atomic

In an *Other-multi-copy atomic* system, it is required that a memory write effect from an Observer, if observed by a different Observer, is then observed by all other Observers that access the Location coherently. It is, however, permitted for an Observer to observe its own writes prior to making them visible to other observers in the system.

The Other-multi-copy atomic property of the Arm memory model is enforced by placing constraints on the possible executions of a program. Those executions that meet the constraints given by the ordering model are said to be Architecturally well-formed. An implementation that is executing a program is only permitted to exhibit behavior consistent with an Architecturally well-formed execution.

Architecturally well-formed

An *Architecturally well-formed* execution must satisfy both the Internal visibility requirement and any of the three alternative External ordering constraints.

B2.3.5 Internal visibility requirement

For a memory read or write effect RW_1 that appears in program order before a memory read or write effect RW_2 to the same Location:

- Where one or more of the following statements is true:
  - RW_1 is not a Tag-Check-read.
  - RW_2 is not a Tag-Check-read.
  - RW_1 and RW_2 are both Tag-Check-reads R_1 and R_2 that are Tag-Location-Ordered.
- The *Internal visibility requirement* requires that exactly one of the following statements is true:
  - RW_2 is a memory write effect W_2 that is Coherence-after RW_1.
  - RW_1 is a memory write effect W_1, RW_2 is a memory read effect R_2 and either:
    - R_2 Reads-from W_1.
    - R_2 Reads-from a memory write effect that is Coherence-after W_1.
  - RW_1 and RW_2 are both reads R_1, R_2, R_1 Reads-from a memory write effect W_3 and either:
    - R_2 Reads-from W_3.
    - R_2 Reads-from a memory write effect that is Coherence-after W_3.

Informally, if a Memory effect M_1 from an Observer appears in program order before a Memory effect M_2 from the same Observer, then M_1 will be seen to occur before M_2 by that Observer.

B2.3.6 External ordering constraints

The Arm memory model offers the following three alternative representations of the *External ordering constraint*:

- External visibility requirement.
- External completion requirement.
- External global completion requirement.

An Architecturally well-formed execution must satisfy both the Internal visibility requirement and one of the three alternative representations in the External ordering constraints.

External visibility requirement

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A memory read or write effect RW_1 is *Ordered-before* a memory read or write effect RW_2 if and only if any of the following cases apply:

- RW_1 is Observed-by a memory read or write effect RW_3 which is generated by the same instruction as RW_2.
- RW_1 is Locally-ordered-before RW_2.
- RW_1 is Pick-locally-ordered-before RW_2.
- RW_1 is Ordered-before a memory read or write effect that is Ordered-before RW_2.

For a memory read or write effect RW_1 from an Observer that is Ordered-before a memory read or write effect RW_2 from a different Observer, the External visibility requirement requires that RW_2 is not Observed-by RW_1. This means that an Architecturally well-formed execution must not exhibit a cycle in the Ordered-before relation.

Informally, if a Memory effect M_1 from an Observer appears in program order before a Memory effect M_2 from the same Observer, then M_1 will be seen to occur before M_2 by all Observers in the system.

Completes-before order

The *Completes-before order* is a total order that corresponds to the order in which Memory effects complete within the system. The following effects constitute a single entry in the Completes-before order:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Completes-before order.

Completes-before

A memory read or write effect RW_1 *Completes-before* a memory read or write effect RW_2 if and only if RW_1 appears in the Completes-before order before RW_2.

Deriving Reads-from and Coherence order from the Completes-before order

The Completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- For a memory read effect R_1 of a memory location by an Observer, then:
  — If there is a memory write effect W_2 to the same Location from the same Observer and all of the following are true:
    — W_2 appears in program order before R_1.
    — R_1 Completes-before W_2.
    — There are no writes to the Location appearing in program order between W_2 and R_1 then R_1 Reads-from W_2.
  — Otherwise, R1 Reads-from its closest preceding write in the Completes-before order to the same Location. If no such write exists, then R1 Reads-from the initial value of the memory location.
- The Coherence order of writes to a memory location is the order in which those writes appear in the Completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

External completion requirement

A memory read or write effect RW_1 Completes-before a memory read or write effect RW_2 if and only if any of the following statements are true:

- RW_1 is Locally-ordered-before RW_2.
- RW_1 is Pick-locally-ordered-before RW_2.
- RW_1 is a memory read effect R_1 and RW_2 is a memory read effect R_2 and R_1 is Single-copy-atomic-ordered-before R_2.

Globally-completes-before order

The Globally-completes-before order is a total order that corresponds to the order in which Memory effects globally-complete within the system. The following effects constitute a single entry in the Globally-completes-before order:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.

- Reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Globally-completes-before order.

Globally-completes-before

A memory read or write effect RW_1 *Globally-completes-before* a memory read or write effect RW_2 if and only if RW_1 appears in the Globally-completes-before order before RW_2.

Deriving Reads-from and Coherence order from the Globally-completes-before order

The Globally-completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- A memory read effect R_1 of a memory location by an Observer Reads-from its closest preceding write in the Globally-completes-before order to the same Location. If no such write exists, then R_1 Reads-from the initial value of the memory location.
- The Coherence order of writes to a memory location is the order in which those writes appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Globally-completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

The *External global completion* requirement requires that a memory read or write effect RW_1 Globally-completes-before a memory read or write effect RW_2 if and only if any of the following statements are true:

- RW_1 is Locally-ordered-before RW_2 and either:
  - RW_1 is a memory write effect.
  - RW_1 is a memory read effect R_1 and either:
    - R_1 is not a Local read successor of a memory write effect.
    - R_1 is a Local read successor of a memory write effect that is Locally-ordered-before RW_2.
    - R_1 is a Local read successor of a memory write effect that is Pick-locally-ordered-before RW_2.
- RW_1 is a memory read effect R_1 and RW_2 is a memory read effect R_2 and R_1 is Single-copy-atomic-ordered-before R_2.

B2.3.7 Completion and endpoint ordering

Interaction between Observers in a system is not restricted to communication via shared variables in coherent memory. For example, an Observer could configure an interrupt controller to raise an interrupt on another Observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different Observers. When these forms of interaction are used in conjunction with shared variables, a `DSB` instruction can be used to enforce ordering between them.

For all memory, the completion rules are defined as:

- A memory read effect R_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain will be Coherence-after R_1.
  - Any translation table walks associated with R_1 are complete for that shareability domain.
- A memory write effect W_1 to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain will be Coherence-after W_1.

  — Any read to the same Location by an Observer within the shareability domain will either Reads-from W_1 or Reads-from a memory write effect that is Coherence-after W_1.
  — Any translation table walks associated with the write are complete for that shareability domain.

- A translation table walk is complete for a shareability domain when the memory accesses, including the updates to translation table entries, associated with the translation table walk are complete for that shareability domain, and the TLB is updated.
- A cache maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

Note

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side-effects is complete only when the read or write both:
- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

Note

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and writes to a peripheral which is defined as the Peripheral coherence order:

Memory-mapped peripheral

A *Memory-mapped peripheral* occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side-effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

Peripheral coherence order

The *Peripheral coherence order* of a Memory-mapped peripheral is a total order on all reads and writes to that peripheral.

Note

The Peripheral coherence order for a Memory-mapped peripheral signifies the order in which accesses arrive at the endpoint.

For a memory read or write effect RW_1 and a memory read or write effect RW_2 to the same peripheral, then RW_1 will appear in the Peripheral coherence order for the peripheral before RW_2 if either of the following cases apply:

- RW_1 and RW_2 are accesses using Non-cacheable or Device attributes and RW_1 is Ordered-before RW_2.

- RW_1 and RW_2 are accesses using Device-nGnRE or Device-nGnRnE attributes, with the same XS attribute value, and RW_1 appears in program order before RW_2.

Note

When FEAT_XS is implemented, if accesses marked with the Device-nGnRE or Device-nGnRnE attributes are within the same Memory-mapped peripheral, but the XS attribute is not the same on those accesses, the order of arrival at the endpoint is not defined by the architecture.

Out-of-band-ordered-before

A memory read or write effect RW_1 is *Out-of-band-ordered-before* a memory read or write effect RW_2 if and only if either of the following cases apply:

- RW_1 appears in program order before a DSB instruction that begins an IMPLEMENTATION DEFINED instruction sequence indirectly leading to the generation of RW_2.
- RW_1 is Ordered-before a memory read or write effect RW_3 and RW_3 is Out-of-band-ordered-before RW_2.

If a Memory effect M_1 is Out-of-band-ordered-before a memory read or write effect M_2, then M_1 is seen to occur before M_2 by all Observers.

Note

Arm expects that, in most systems with early acknowledgments, those acknowledgments will come from a point at or after the point that establishes global visibility. This is expected in such systems to enable the acknowledgments to be used as part of the mechanisms to implement the ordering requirements of the Arm memory model.

B2.3.8 SVE memory ordering relaxations

I_{CTNGV} The Arm memory model is relaxed for reads and writes generated by SVE load and store instructions.

R_{QLJPC} When two reads generated by SVE vector load instructions have an address dependency, the dependency does not contribute to the dependency-ordered-before relation.

R_{YMBMZ} When a pair of reads access the same location, and at least one of the reads is generated by an SVE load instruction, for a given observer, the pair of reads is not required to satisfy the internal visibility requirement.

R_{CJHWV} When a single SVE vector store instruction generates multiple writes to the same location, the instruction ensures that these writes appear in the coherence order for that location, in order of increasing vector element number. No other ordering restrictions apply to memory effects generated by the same SVE store instruction.

R_{LVXTJ} If a single SVE load instruction generates multiple reads, the order in which the reads for different elements and registers appear is not architecturally defined.

R_{VMDYZ} If an address dependency exists between two memory reads and an SVE non-temporal vector load instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

I_{CCWGN} For any SVE load or store instruction that generates multiple single-copy atomic accesses to Normal or Device memory, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic memory element access sizes.

B2.3.9 Ordering of instruction fetches

For two memory locations A and B, if A has been written to with an updated value and been made coherent with the instruction fetches of the shareability domain before B has been written to with an updated value by an observer in the same shareability domain, then where, for an observer in the shareability domain, an instruction read from B appears in program order before an instruction fetched from A, if the instruction read from B contains the updated value of B then the instruction read from A appearing later in program order will contain the updated value of A.

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1}

The write is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1}

The write is complete for the shareability domain.

——— **Note** ———

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When CTR_EL0.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the `IC IVAU` instructions of a different core.
  - When CTR_EL0.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

———————

B2.3.10 Restrictions on the effects of speculation

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a Translation fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.
- When data is loaded under speculation from a location without a translation for the translation regime being speculated in, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.

- When data is loaded as a result of speculative accesses made after `TLBI` + `DSB` + `ERET` using a translation that was invalidated by the TLBI, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.
- Changes to Special-purpose registers can occur speculatively.
- Execute-never controls apply to speculative instruction fetching. See *Access permissions for instruction execution* on page D5-4855.
- When writing new instructions to memory, there is no requirement for an `SB` instruction to prevent speculative execution of the old code. See *Instruction cache maintenance instructions* on page K10-11355.

Note

The prohibition of using data loaded under speculation with faults to form addresses, condition codes or SVE predicate values does not prohibit the use of value predicted data from such locations for such purposes, so long as the training of the data value prediction was from the hardware defined context that is using the prediction. A consequence of this is that training of value prediction cannot be based on data loaded under speculation with a translation or Permission fault.

Speculative Store Bypass Safe (SSBS)

When FEAT_SSBS is implemented, PSTATE.SSBS is a control that can be set by software to indicate whether hardware is permitted to use, in a manner that is potentially speculatively exploitable, a speculative value in a register that has been loaded from memory using a load instruction that speculatively read the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

When the value of PSTATE.SSBS is 0, hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Note

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.
- Software written for architectures from Armv8.0 to Armv8.4 will set SPSR_ELx.SSBS to 0. This means that PSTATE.SSBS will not set, so hardware will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

Restrictions on exploitative control of speculative execution

The execution of some code (code1) can exploitatively control speculative execution of some other code (code2) if and only if all of the following apply:

- The actions of code1 can influence the speculative execution of code2 to cause an irreversible change to the microarchitectural state of the PE that is indicative of some architectural state accessible to the execution context of code2.
- Code1 has control in determining the choice of the architecture state that causes the irreversible change to the microarchitectural state.
- The irreversible changes to the microarchitectural state of the PE can be measured by code executing in an execution context other than that of code2 to allow the retrieval of the architectural state in a computationally feasible manner.

Restrictions on the effects of speculation from Armv8.5

Note

If SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated by software for:
- Secure EL0 for all ASID and VMID values.
- Secure EL1 for all VMID values.

From Armv8.5, there are some further restrictions on the effects of speculation in addition to those in Armv8.0:

- Data loaded under speculation with a Permission or Domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.
- Any System register read under speculation to a register that is not architecturally accessible from the current Exception level cannot be used to form an address, to generate condition codes, or to generate SVE predicate values to be used by other instructions in the speculative sequence.

  Note

  As the effects of speculation are not architecturally visible, this restriction requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- Code running in one hardware-defined context cannot exploitatively control speculative execution of code in a different hardware-defined context as a result of the behavior of any execution prediction resources that predict address or register values. In the case of this definition, the hardware-defined context is determined by:
  - The Exception level.
  - The Security state.
  - When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
  - When executing at EL0 and using the EL1&0 translation regime, the *address space identifier* (*ASID*) and, if EL2 is implemented and enabled in the current Security state, the VMID.
  - When executing at EL0 and using the EL2&0 translation regime, the ASID.
  - When in AArch64 state, the current SCXTNUM_ELx value if SCXTNUM_ELx is implemented and the hardware identifies that SCXTNUM_ELx is part of the context. Where SCXTNUM_ELx is not included as part of the hardware-indicated context, an implementation can further identify that branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

Note

— The definition of "hard-to-determine manner" is left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.

— The architecture does not require that prediction resources that simply predict the direction of a branch are separated in this way.

- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.
- Changes to Special-purpose registers can occur speculatively.

B2.3.11 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the Arm architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The following subsections describe the Arm memory barrier instructions:

- *Instruction Synchronization Barrier (ISB)*
- *Data Memory Barrier (DMB)* on page B2-173.
- *Data Synchronization Barrier (DSB)* on page B2-175.
- *Speculation Barrier (SB)* on page B2-173.
- *Consumption of Speculative Data Barrier (CSDB)* on page B2-174.
- *Speculative Store Bypass Barrier (SSBB)* on page B2-174.
- *Profiling Synchronization Barrier (PSB CSYNC)* on page B2-174.
- *Physical Speculative Store Bypass Barrier (PSSBB)* on page B2-174.
- *Trace Synchronization Barrier (TSB CSYNC)* on page B2-175
- *Shareability and access limitations on the data barrier operations* on page B2-176.
- *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-177.
- *LoadLOAcquire, StoreLORelease* on page B2-178.

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

`DMB` and `DSB` instructions affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

Instruction Synchronization Barrier (ISB)

An `ISB` instruction ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed. Using an `ISB` ensures that the effects of context-changing operations executed before the `ISB` are visible to the instructions fetched after the `ISB` instruction. Examples of context-changing operations that require the insertion of an `ISB` instruction to ensure the effects of the operation are visible to instructions fetched after the `ISB` instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the `ISB` instruction take effect only after the `ISB` has been executed.

The pseudocode function for the operation of an `ISB` is `InstructionSynchronizationBarrier()`.

See also *Memory barriers* on page D4-4765.

Data Memory Barrier (DMB)

The `DMB` instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The `DMB` instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the `DMB` instruction is covered formally in the *Definition of the Arm memory model* on page B2-157 and this introduction to the `DMB` instruction is not intended to contradict that section.

The basic principle of a `DMB` instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the `DMB` instruction. The `DMB` instruction ensures that all affected memory accesses by the PE executing the `DMB` instruction that appear in program order before the `DMB` instruction and those which originate from a different PE, to the extent required by the DMB options, which have been Observed-by the PE before the `DMB` instruction is executed, are Observed-by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the `DMB` instruction are Observed-by that PE.

The use of a `DMB` instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The `DMB` instruction affects only memory accesses and the operation of data cache and unified cache maintenance instructions, see *A64 Cache maintenance instructions* on page D4-4742. It has no effect on the ordering of any other instructions executing on the PE.

The pseudocode function for the operation of a `DMB` instruction is `DataMemoryBarrier()`.

Speculation Barrier (SB)

An `SB` instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier:

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

Speculative execution of an `SB` instruction:
- Cannot be as a result of control flow speculation.
- Cannot be as a result of data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An `SB` instruction can complete when:
- It is known that it is not speculative.
- All the predicted data values generated by instructions appearing in program order before the `SB` instruction have their predicted values confirmed.

Note

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

Consumption of Speculative Data Barrier (CSDB)

The `CSDB` instruction is a memory barrier instruction that controls speculative execution and data value prediction. This includes:

- Data value predictions of any instructions.
- `PSTATE.{N,Z,C,V}` predictions of any instructions other than conditional branch instructions appearing in program order before the `CSDB` that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

For purposes of the definition of CSDB, `PSTATE.{N,Z,C,V}` is not considered a data value. This definition permits:

- Control flow speculation before and after the `CSDB` instruction.
- Speculative execution of conditional data processing instructions after the CSDB instruction, unless they use the results of data value or `PSTATE.{N,Z,C,V}` predictions of instructions appearing in program order before the `CSDB` instruction that have not been architecturally resolved.

Speculative Store Bypass Barrier (SSBB)

The `SSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `SSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the `SSBB` instruction.
- When a load to a location appears in program order before the `SSBB` instruction, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the `SSBB` instruction.

Profiling Synchronization Barrier (PSB CSYNC)

The `PSB CSYNC` instruction is a memory barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a `NOP`.

Physical Speculative Store Bypass Barrier (PSSBB)

The `PSSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `PSSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the `PSSBB` instruction.

- When a load to a location appears in program order before the `PSSBB` instruction, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - — The store is to the same location as the load.
  - — The store appears in program order after the `PSSBB` instruction.

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

Trace Synchronization Barrier (TSB CSYNC)

The `TSB CSYNC` instruction is a memory barrier instruction that preserves the relative order of memory accesses to System registers due to trace operations and other memory accesses to the same registers.

A trace operation is an operation of the PE Trace Unit generating trace for an instruction when FEAT_TRF is implemented and enabled.

A `TSB CSYNC` instruction is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that `TSB CSYNC` instruction is executed in the necessary order.

If trace is generated between a Context synchronization event and a `TSB CSYNC` operation, these trace operations may be reordered with respect to the `TSB CSYNC` operation, and therefore may not be synchronized.

The following situations are synchronized using a `TSB CSYNC` operation:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - — A is executed in program order before a Context synchronization event C.
  - — C is in program order before a `TSB CSYNC` operation T.
  - — B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - — A is executed in program order before a Context synchronization event C1.
  - — C1 is in program order before `TSB CSYNC` operation T.
  - — T is executed in program order before a second Context synchronization event C2.
  - — B is executed in program order after C2.

A `TSB CSYNC` operation is not needed to ensure a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a Context synchronization event C.
- B is executed in program order before C.

The pseudocode function for the operation of a `TSB CSYNC` instruction is `TraceSynchronizationBarrier()`.

Data Synchronization Barrier (DSB)

A `DSB` instruction is a memory barrier that ensures that memory accesses that occur before the `DSB` instruction have completed before the completion of the `DSB` instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a `DSB` instruction:

- At EL2 ensures that any memory accesses caused by Speculative translation table walks from the EL1&0 translation regime have been observed.
- At EL3 ensures that any memory accesses caused by speculative translation table walks from the EL2, EL1&0 or EL2&0 translation regimes have been observed.

For more information, see *Use of out-of-context translation regimes* on page D5-4792.

A DSB instruction executed by a PE, PEe, completes when all of the following apply:

- All explicit memory effects of the required access types appearing in program order before the DSB are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, the following instructions issued by PEe before the DSB are complete for the required shareability domain:
  - All cache maintenance instructions.
  - All TLB maintenance instructions.
  - All PSB CYNC instructions.
- When FEAT_XS is implemented, if the required access types of the DSB is reads and writes, completion of the DSB instruction with the nXS qualifier executed by a PE, PEe, ensures that:
  - All previous TLBInXS maintenance operations generated by AArch64 TLB maintenance instructions with the nXS qualifier executed by PEe are finished for all PEs in the shareability domain of the DSB instruction.
  - All previous TLBInXS maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 by PEe when HCRX_EL2.FnXS is 1 are finished for all PEs in the shareability domain of the DSB instruction.

  Completion of the DSB instruction with the nXS qualifier executed by a PE, PEe, does not ensure that:
  - All previous TLB maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 by PEe when HCRX_EL2.FnXS is 0 are finished for all PEs in the shareability domain of the DSB instruction.
  - All previous TLB maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL2 or EL3 by PEe are finished for all PEs in the shareability domain of the DSB instruction.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes other than:

- Being fetched from memory and decoded.
- Reading the general-purpose, SIMD and floating-point, SVE vector or predicate, Special-purpose, or System registers that are directly or indirectly read without causing side-effects.
- If FEAT_ETS is not implemented, having any virtual addresses of loads and stores translated.

If FEAT_MTE2 is implemented, on completion of a DSB instruction operating over the Non-shareable domain, all updates to TFSR_ELx.TFx or TFSRE0_EL1.TFx due to Tag Check fails caused by accesses for which the DSB operates will be complete. For more information on FEAT_MTE2, see Chapter D6 *The Memory Tagging Extension*.

When FEAT_XS is implemented and HCRX_EL2.FnXS is 1, an AArch64 DSB instruction executed at EL1 or EL0 behaves in the same way as the corresponding DSB instruction with the nXS qualifier executed at EL1 or EL0.

The pseudocode function for the operation of a DSB is `DataSynchronizationBarrier()`.

See also:
- *Memory barriers* on page D4-4765.
- *Ordering and completion of TLB maintenance instructions* on page D5-4927.

Shareability and access limitations on the data barrier operations

The DMB and DSB instructions take an argument that specifies:
- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.

  - Non-shareable.

  Full system applies to all the observers in the system and, as such, encompasses the Inner and Outer Shareable domains of the processor.

  Note

  The distinction between Full system and Outer Shareable is applicable only for Normal Non-cacheable memory accesses and Device memory accesses.

- The accesses for which the instruction operates. This is one of:
  - Read and write accesses, both before and after the barrier instruction.
  - Write accesses only, before and after the barrier instruction.
  - Read accesses before the barrier instruction, and read and write accesses after the barrier instruction.

    Note

    This form of a `DMB` or `DSB` instruction can be described as a load-load/store barrier.

For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page B2-173 or *Data Synchronization Barrier (DSB)* on page B2-175.

Table B2-1 on page B2-177 shows how these options are encoded in the `<option>` field of the instruction:

Table B2-1 Encoding of the `DMB` and `DSB` `<option>` parameter

Accesses		**Shareability domain**			
Before the barrier	**After the barrier**	**Full system**	**Outer Shareable**	**Inner Shareable**	**Non-shareable**
Reads and writes	Reads and writes	`SY`	`OSH`	`ISH`	`NSH`
Writes	Writes	`ST`	`OSHST`	`ISHST`	`NSHST`
Reads	Reads and writes	`LD`	`OSHLD`	`ISHLD`	`NSHLD`

See the instruction descriptions for more information:

- *DMB* on page C6-1461.
- *DSB* on page C6-1464.

Note

`ISB` also supports an optional limitation argument that can contain only one value that corresponds to full system operation, see *ISB* on page C6-1487.

Load-Acquire, Load-AcquirePC, and Store-Release

Arm provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the *Release Consistency sequentially consistent* (RCsc) model. In addition, FEAT_LRCPC provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release can be use to support the weaker *Release Consistency processor consistent* (RCpc) model.

The full definitions of the Load-Acquire and Load-AcquirePC instructions are covered formally in the *Definition of the Arm memory model* on page B2-157. This introduction to the Load-Acquire and Load-AcquirePC instructions is not intended to contradict that section.

The basic principle of both Load-Acquire and Load-AcquirePC instructions is to introduce order between:

- The memory access generated by the Load-Acquire or Load-AcquirePC instruction.

- The memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction, such that the memory access generated by the Load-Acquire or Load-AcquirePC instruction is Observed-by each PE to the extent that the PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction are Observed-by that PE to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire or Load-AcquirePC instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the *Definition of the Arm memory model* on page B2-157 and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the following:

- A set of memory accesses, RWx, that are generated by the PE executing the Store-Release instruction and that appear in program order before the Store-Release instruction, together with those that originate from a different PE to the extent that the PE is required to observe them coherently, Observed-by the PE before executing the Store-release.
- The memory access generated by the Store-Release (Wrel), such that all of the memory accesses, RWx, are Observed-by each PE to the extent that the PE is required to observe those accesses coherently, before Wrel is Observed-by that PE to the extent that the PE is required to observe that access coherently.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Where a Load-Acquire appears in program order after a Store-Release, the memory access generated by the Store-Release instruction is Observed-by each PE to the extent that PE is required to observe the access coherently, before the memory access generated by the Load-Acquire instruction is Observed-by that PE, to the extent that the PE is required to observe the access coherently. In addition, the use of a Load-Acquire, Load-AcquirePC or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral coherence order.

Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements. The address supplied to the instructions must be aligned to twice the size of the element being loaded, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit `DMB` instruction.

LoadLOAcquire, StoreLORelease

For each PE, the Non-secure physical memory map is divided into a set of LORegions using a table that is held within the PE. Any PA in the Non-secure memory map can be a member of one LORegion. If a PA is assigned to more than one LORegion, then an implementation might treat it as if it has been assigned to fewer LORegions than that have been specified. A PA in the Secure physical memory map cannot be a member of any LORegion. For more information, see *Limited ordering regions* on page B2-179.

Armv8.1 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores that apply in relation to the defined LORegions. The new variants of the Load-Acquire and Store-Release instructions are LoadLOAcquire and StoreLORelease. See *LoadLOAcquire/StoreLORelease* on page C3-259.

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

- LoadLOAcquire has the same semantics as Load-Acquire except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-177.
- StoreLORelease has the same semantics as Store-Release except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-177.

In addition, for accesses to Memory-mapped peripherals:

- LoadLOAcquire has the same semantics as Load-Acquire except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-177.
- StoreLORelease has the same semantics as Store-Release except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-177.

Note

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit `DMB` instruction.

B2.3.12 Limited ordering regions

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

This feature is supported in AArch64 state only.

Specification of the LORegions

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

Each LORegion descriptor consists of:

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

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

The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1, and LORC_EL1 registers in the System register space. These registers describe only memory addresses in the Non-secure memory map. These registers are UNDEFINED if accessed when SCR_EL3.NS == 0.

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

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

Note

If no LORegions are implemented, then the LoadLOAcquire and StoreLORelease will therefore behave as a Load-Acquire and Store-Release.

A new access type `AccType_LIMITEDORDERED` has been added for these limited ordering instructions to be identified.

B2.4 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The Arm architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- *Introduction to caches*.
- *Memory hierarchy*.
- *Application level access to functionality related to caches* on page B2-182
- *Implication of caches for the application programmer* on page B2-183.
- *Preloading caches* on page B2-185.

B2.4.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

Spatial locality

An access to one Location is likely to be followed by accesses to adjacent Locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The Arm architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B2.4.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B2-1 on page B2-182 shows an example of such a system in an Armv8-A system that supports virtual addressing.

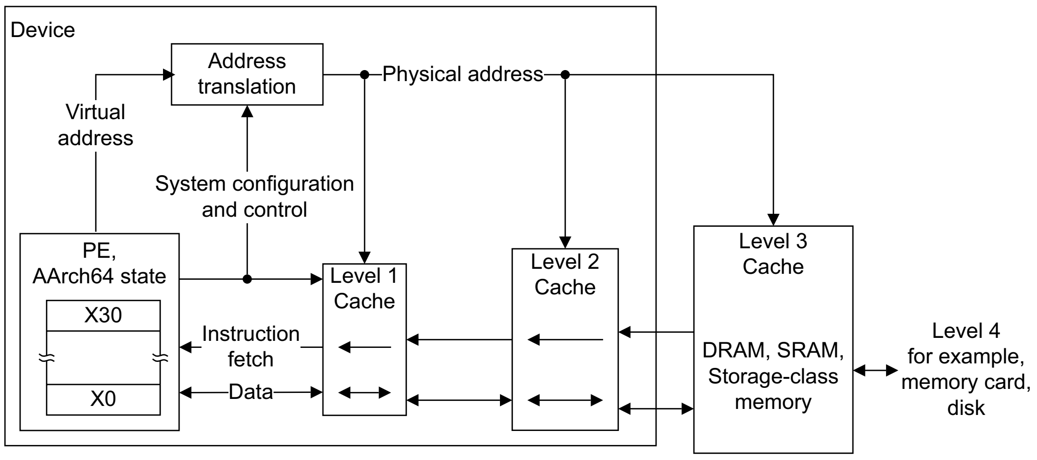


Figure B2-1 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU), Point of Coherency (PoC), Point of Persistence (PoP), and Point of Deep Persistence (PoDP).

For more information, including the definitions of PoU, PoC, PoP, and PoDP, see *About cache maintenance in AArch64 state* on page D4-4738.

If FEAT_MTE2 is implemented, the behavior of cache maintenance instructions is modified. For more information, see *Allocation Tags* on page D6-4937.

The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about Cacheability and Shareability, see *Memory types and attributes* on page B2-193.

B2.4.3 Application level access to functionality related to caches

As indicated in *About the Application level programmers' model* on page B1-134, the application level corresponds to execution at EL0. The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable use of some of this functionality from EL0, as follows:

When the value of SCTLR_EL1.UCI is 1

Software executing at EL0 can access:

- The data cache maintenance instructions, `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CVADP`, and `DC CIVAC`. See *The data cache maintenance instruction (DC)* on page D4-4744.

- The instruction cache maintenance instruction `IC IVAU`. See *The instruction cache maintenance instruction (IC)* on page D4-4744.

Attempted execution of these instructions might generate a Permission fault as described in *Permission fault* on page D5-4896.

When the value of SCTLR_EL1.UCT is 1

Software executing at EL0 can access the cache type register. See CTR_EL0.

When the value of SCTLR_EL1.DZE is 1

Software executing at EL0 can access the data cache zero instruction `DC ZVA`. See *Data cache zero instruction* on page D4-4756.

The SCTLR_EL1.{UCI, UCT, DZE} control fields are accessible only by software executing at EL1 or higher.

When HCR_EL2.{E2H, TGE} == 1 the controls {UCI, UCT and DZE} are found in SCTLR_EL2.

This functionality is UNDEFINED at EL0 when the value of the corresponding SCTLR_EL1 control field is 0.

B2.4.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using cache maintenance instructions to manage the coherency issues in software. See *Application level access to functionality related to caches* on page B2-182.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see *Non-shareable Normal memory* on page B2-195 and *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page B2-194.

  Note

  The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations, the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Note

Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last *Context synchronization event* on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a *Context synchronization event*, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture requires the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory. A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 0}

The location written to has been cleaned to the *Point of unification (PoU)* from the data cache, and that clean is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1}

The write is complete for the shareability domain. Subsequently the location has been invalidated to the *Point of unification (PoU)* from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1}

The write is complete for the shareability domain.

If software requires coherency between instruction execution and memory, it must manage this coherency using *Context synchronization event*s and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```

Note

- If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.

- For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the Armv8-A AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See *Non-cacheable accesses and instruction caches* on page D4-4737.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See *Concurrent modification and execution of instructions* on page B2-154.
- The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.

B2.4.5 Preloading caches

The Arm architecture provides memory system hints `PRFM`, `LDNP`, and `STNP` that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is taken using an SError interrupt exception. For more information, see *ISS encoding for an exception from a Data Abort* on page D13-5373.

`PrefetchHint{}` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PRFM` and load/store instructions that provide hints to the memory system, see *Prefetch memory* on page C3-264 and *Load/store SIMD and floating-point non-temporal pair* on page C3-261. For more information on SVE `PRF*` instructions that provide hints to the memory system, see *Predicated non-contiguous element accesses* on page C3-285.

B2.5 Alignment support

This section describes alignment support. It contains the following subsections:

- *Instruction alignment*.
- *Alignment of data accesses*.

B2.5.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a PC alignment fault. See *PC alignment checking* on page D1-4631.

B2.5.2 Alignment of data accesses

An unaligned access to any type of Device memory causes an Alignment fault.

Unaligned accesses to Normal memory

The behavior of unaligned accesses to Normal memory is dependent on all of the following:

- The instruction causing the memory access.
- The memory attributes of the accessed memory.
- The value of SCTLR_ELx.{A, nAA}.
- Whether or not FEAT_LSE2 is implemented.

Load or Store of Single or Multiple registers

For all instructions that load or store single or multiple registers, but not Load-Exclusive, Store-Exclusive, Load-Acquire/Store-Release, Atomic, and `SETG*` Memory Copy and Memory Set instructions, if the address that is accessed is not aligned to the size of the data element being accessed, then:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

- An unaligned access is performed.
- If FEAT_LSE2 is not implemented, the access is not guaranteed to be single-copy atomic except at the byte access level.
- If FEAT_LSE2 is implemented:
  - — If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, the memory access is single-copy atomic. For `LDNP`, `LDP`, or `STP` instructions, the entire memory access will be single-copy atomic.
  - — If all the bytes of the memory accessed do not lie within a 16-byte quantity aligned to 16 bytes, or the access is not to Normal Inner Write-Back, Outer Write-Back Cacheable memory, the access is not guaranteed to be single-copy atomic except at the byte access level.

For these instructions, the definition of an unaligned access is based on the size of the accessed elements, not the overall size of the memory access. This affects SIMD and SVE element and structure loads and stores, and also load/store pair instructions.

For predicated SVE vector element and structure load or store instructions, alignment checks are based on the memory element access size, not on the vector element size.

For predicated SVE vector element and structure load or store instructions, *Inactive elements* cannot cause an Alignment fault.

For unpredicated SVE vector register load or store instructions, the base address is checked for 16-byte alignment.

For unpredicated SVE predicate register load or store instructions, the base address is checked for 2-byte alignment.

Load-Exclusive/ Store-Exclusive and Atomic instructions

For Load-Exclusive/Store-Exclusive, and Atomic instructions including those with acquire or acquire-release semantics:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, these instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

If FEAT_LSE2 is implemented, then:

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, an unaligned access is performed.
- If all the bytes of the memory access do not lie within a 16-byte quantity aligned to 16-bytes, or the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.

Where memory access is performed, then it is single-copy atomic.

For these instructions, the definition of an unaligned access is based on the overall access size.

If FEAT_LS64 is implemented, when a single-copy atomic 64-byte instruction accesses a memory location that is not aligned to 64 bytes, an Alignment fault always occurs, regardless of the value of SCTLR_ELx.A.

Non-atomic Load-Acquire/Store-Release instructions

For Load-Acquire/Store-Release instructions that do not have exclusive or atomic behaviors:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, then these instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

If FEAT_LSE2 is implemented, then:

- If the memory access is not to Normal Inner Write-Back or Outer Write-Back Cacheable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.
- If all of the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes then the following applies:
  - If SCTLR_ELx.nAA applicable to the current Exception level is 0, an Alignment fault is generated.
  - If SCTLR_ELx.nAA applicable to the current Exception level is 1, then an unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.

  In this case, the architecture does no define the order of the different transactions of the access defined by the single instructions relative to each other.
- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, an unaligned access meeting all the semantics of the instruction is performed.

Note

- Unaligned accesses typically take additional cycles to complete compared to a naturally-aligned access.
- An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary.

Memory Copy and Memory Set instructions

For `SETG*` instructions:

- There is an alignment check regardless of the value of SCTLR_ELx.A.
- If Xn is not a multiple of 16, an Alignment fault is generated.
- If Xd is not aligned to a multiple of 16, an Alignment fault is generated.

For more information, see the individual `SETG*` instruction descriptions.

B2.6 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- *Instruction endianness* on page B2-190.
- *Data endianness* on page B2-190.
- *Endianness of memory-mapped peripherals* on page B2-192.

B2.6.1 General description of endianness in the Arm architecture

This section describes only memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The quadword at address A.
- The doubleword at address A and A+8.
- The words at addresses A, A+4, A+8, and A+12.
- The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B2-2 have the following definitions:

B_A	Byte at address A.
HW_A	Halfword at address A.
MSByte	Most significant byte.
LSByte	Least significant byte.

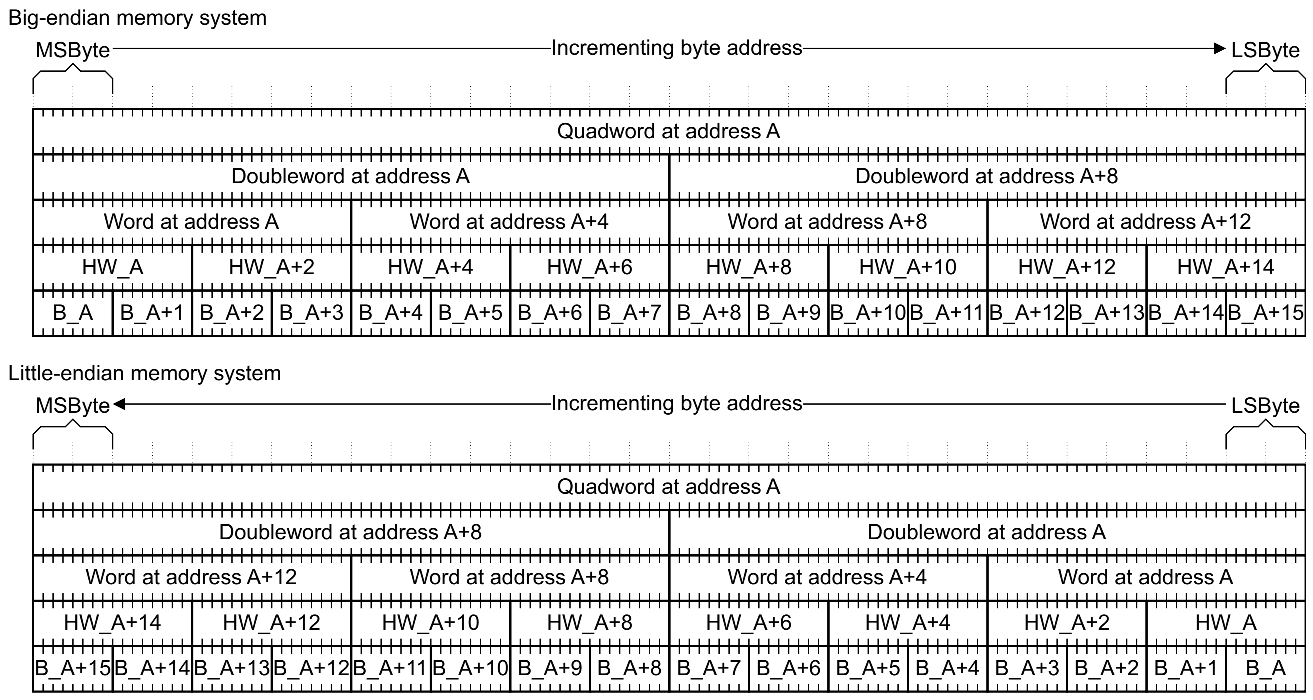


Figure B2-2 Endianness relationships

The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word, or halfword are interpreted. For example, a load of a word from address `0x1000` always results in an access to the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these 4 bytes.

B2.6.2 Instruction endianness

A64 instructions have a fixed length of 32 bits and are always little-endian.

B2.6.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores. For more information, see *Endianness in SIMD operations* on page B2-191.

Note

This means the Armv8 architecture introduces a requirement for 128-bit endian conversions.

Instructions to reverse bytes in a general-purpose register, a SIMD and floating-point register, or an SVE register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table B2-2 on page B2-190 shows the instructions that provide this functionality:

Table B2-2 Byte reversal instructions

Function	Instructions	Notes
Reverse bytes in 32-bit word or words[a]	REV32	For use with general-purpose registers
Reverse bytes in whole register	REV	For use with general-purpose registers
Reverse bytes in 16-bit halfwords	REV16	For use with general-purpose registers
Reverse elements in doublewords, vector	REV64	For use with SIMD and floating-point registers
Reverse elements in words, vector	REV32	For use with SIMD and floating-point registers
Reverse elements in halfwords, vector	REV16	For use with SIMD and floating-point registers
Reverse bytes/halfwords/words within elements, predicated	REVB, REVH, REVW	For use with SVE registers

a. Can operate on multiple words.

Endianness in SIMD operations

SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

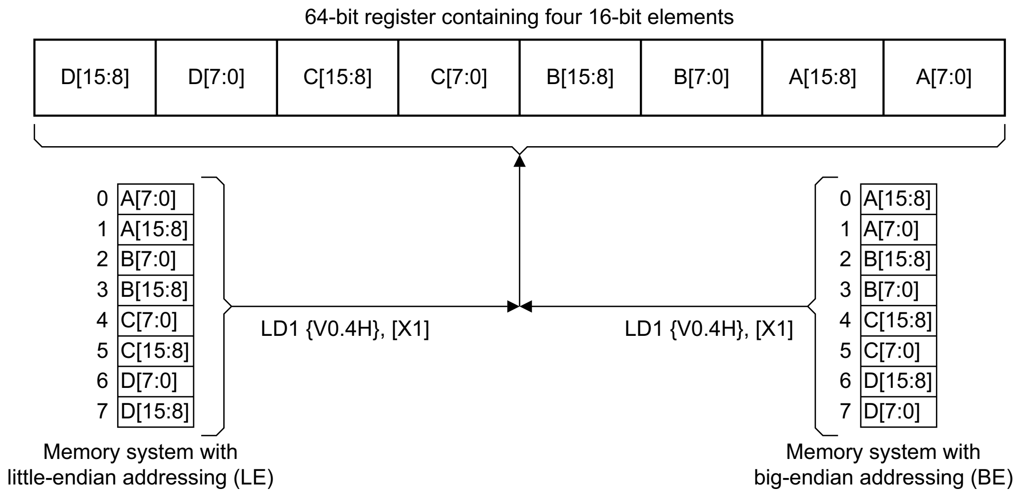


Figure B2-3 SIMD byte order example

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 *Armv8 Pseudocode*.

Endianness in SVE operations

R_{VDGQK} Rules on byte and element order of SIMD load and store instructions apply to SVE load and store instructions.

I_{RFQJP} Additional rules apply to the data endianness of memory accesses performed by SVE load and store instructions.

R_{CNKCL} For predicated SVE vector element and structure load and store instructions, an endianness conversion is performed using the memory element access size. The size of the vector element is not used in endianness conversion.

R_{QHXPL} For unpredicated SVE vector register load and store instructions, the vector byte elements are transferred in increasing element number order without any endianness conversion.

R_{RWLXY} For unpredicated SVE predicate register load and store instructions, each 8 bits from the predicate are transferred as a byte in increasing element number order without any endianness conversion.

R_{YGSBQ} When an SVE load instruction is executed, endianness conversion occurs before any sign-extension or zero-extension into a vector element.

R_{KYRQW} When an SVE store instruction is executed, endianness conversion occurs after any truncation from the vector element to the memory element access size.

B2.6.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a Cross Trigger Interface, see Chapter H8 *About the External Debug Registers*.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see Chapter I2 *System Level Implementation of the Generic Timer*.
- A memory-mapped register interface to the Performance Monitors, see Chapter I3 *Recommended External Interface to the Performance Monitors*.
- A memory-mapped register interface to the Activity Monitors, see Chapter I4 *Recommended External Interface to the Activity Monitors*.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *ARM® Embedded Trace Macrocell Architecture Specification, ETMv4.*

B2.7 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- *Normal memory*.
- *Device memory* on page B2-197.
- *Memory access restrictions* on page B2-204.

B2.7.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform *Speculative* data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time.
- Writes to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See *Multi-register loads and stores that access Normal memory* on page B2-197.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in *Atomicity in the Arm architecture* on page B2-152 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.

The following sections describe the other attributes for Normal memory:

- *Shareable Normal memory* on page B2-194.

- *Non-shareable Normal memory* on page B2-195.
- *Cacheability attributes for Normal memory* on page B2-195.

See also:

- *Multi-register loads and stores that access Normal memory* on page B2-197.
- *Atomicity in the Arm architecture* on page B2-152.
- *Memory barriers* on page B2-172.
- *Concurrent modification and execution of instructions* on page B2-154.

Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

- Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
- Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see *Synchronization and coherency issues between data and instruction accesses* on page B2-184.

Note

- System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.

- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is a member of only a single Inner Shareability domain.
- Each observer is a member of only a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.

- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

The details of the use of the shareability attributes are system-specific. Example B2-1 on page B2-195 shows how they might be used.

Example B2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see *Cacheability, cache allocation hints, and cache transient hints* on page D4-4734.

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint, this does not affect the behavior of accesses to that location.
- For other cases, see *Mismatched memory attributes* on page B2-205.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a `DMB` instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see *Ordering relations* on page B2-161.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example B2-2, Example B2-3 on page B2-197, and Example B2-4 on page B2-197 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see *Memory hierarchy* on page B2-181.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example B2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.

Example B2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example B2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more SVE or SIMD&FP registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B2.7.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering Identified as G or nG, see *Gathering* on page B2-200.

Reordering Identified as R or nR, see *Reordering* on page B2-201.

Early Write Acknowledgement
Identified as E or nE, see *Early Write Acknowledgement* on page B2-202.

The Armv8 Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early Write Acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that *Speculative* accesses to Device-GRE memory is forbidden.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  The following exceptions to this apply:
  - Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.
  - For reads, including hardware speculation, that are performed by an SVE unpredicated load instruction, all of the following are true:
    - For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction, any byte in the window can be accessed by the instruction.
    - All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction.
  - For reads, including hardware speculation, that are performed by an SVE predicated load instruction that is not a non-temporal load, all of the following are true:
    - For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.
    - All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.
  - For reads, including hardware speculation, that are performed by an SVE predicated non-temporal load instruction from memory locations with the Gathering attributes, all of the following are true:
    - For any 128-byte window aligned to 128 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.
    - All bytes accessed by the instruction are in a 128-byte window aligned to 128 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.
  - The architecture permits a Memory Copy and Memory Set `CPY*` instruction to perform speculative reads of any memory location, even those marked as Device, within a 64-byte quantity, aligned to 64 bytes, of a location that is within the range [Xs] to [Xs+Xn-1].
  - For Device memory with the Gathering attribute, reads generated by the `LDNP` instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.
  - Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See *Properties of single-copy atomic accesses* on page B2-154.

Note

- An instruction that generates a sequence of accesses as described in *Atomicity in the Arm architecture* on page B2-152 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception, the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.
- Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit memory write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two writes of Device memory type to the same location might be merged before they reach the endpoint, unless both writes have the non-Gathering attribute or there is an ordered-before relationship between the two writes.
- For accesses to any Device memory type, a `DMB` instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of implementation defined size. For more information, see *Ordering relations* on page B2-161.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

  Note

  This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note

In the EL1&0 translation regime in systems where HCR_EL2.TGE==1 and HCR_EL2.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

See also *Memory access restrictions* on page B2-204.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

Note

In general, making a translation table walk to any Device memory type is the result of a programming error.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

Gathering

In the Device memory attribute:

G	Indicates that the location has the Gathering attribute.
nG	Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

Note

This also applies to writebacks from the cache, whether caused by a *Natural eviction* or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their single-copy atomic sizes, except that there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by multi-register load/store instructions that generate more than one single-copy atomic access. See *Multi-register loads and stores that access Device memory* on page B2-203.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

Gathering between memory accesses generated by Memory Copy and Memory Set instructions is always permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.

— The accesses are not generated by a Store-Release instruction.

- The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Reordering

In the Device memory attribute:

R	Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see *Ordering relations* on page B2-161.
nR	Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

——— Note ———

Some interconnect fabrics, such as PCIe, perform very limited reordering, which is not important for the software usage. It is outside the scope of the Arm architecture to prohibit the use of a non-Reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

——— Note ———

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the `DMB` instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.
- The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an IMPLEMENTATION DEFINED size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in *Ordering relations* on page B2-161 as part of the formal definition of the memory model.
- If the same memory location is mapped with different aliases, and different attribute values, these are a type of mismatched attribute. The different attributes could be:
  — A different Reordering attribute value.
  — A different Device memory attribute value.
  — When FEAT_XS is implemented, a different XS attribute value.

  For information about the effects of accessing memory with mismatched attributes, see *Mismatched memory attributes* on page B2-205.

An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

The non-Reordering attribute has no effect on the memory accesses caused by Memory Copy and Memory Set instructions.

Early Write Acknowledgement

In the Device memory attribute:

E	Indicates that the location has the Early Write Acknowledgement attribute.
nE	Indicates that the location has the No Early Write Acknowledgement attribute.

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgement of a write comes from the endpoint, it is guaranteed that:
  - — Only the endpoint of the write access returns a write acknowledgement of the access.
  - — No earlier point in the memory system returns a write acknowledgement.
- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgement of a write comes from the endpoint, the acknowledgement of a write is not required to come from the endpoint.

Note

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgement of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgement is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.

- The areas of the physical memory map for which write acknowledgement from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgement from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgement from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgement from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgement from the endpoint.

Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register, the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

For all instructions that load or store one or more SIMD&FP or SVE registers, and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions, even for access to any type of Device memory.

SVE loads and stores that access Device memory

R_{XLLJZ} All rules applying to Device memory accesses by Advanced SIMD and floating-point load and store instructions apply to Device memory access by SVE load and store instructions.

I_{YHWJT} Additional rules apply to Device memory access by SVE load and store instructions.

R_{NYMWH} When an SVE vector prefetch instruction is executed, any resulting memory read is guaranteed not to access Device memory.

R_{SBHLD} When an SVE Non-fault vector load is executed, or when any element from a First-fault load except the *First active element* attempts to access Device memory, the resulting memory reads will not access Device memory.

R_{TMVNR} When an SVE Non-fault vector load instruction is executed, an attempt by any *Active element* to access Device memory is suppressed and reported in the FFR.

R_{SFBKQ} When an SVE First-fault vector load instruction is executed, any memory read performed for the *First active element* can access Device memory.

R_{BHNQN} When an SVE First-fault vector load instruction is executed, an attempt by any *Active element* other than the *First active element* to access Device memory is suppressed and is reported in the FFR.

R_{QBLMZ} Any access to Device memory performed by an SVE load or store instruction is relaxed such that it might behave as if:

- The Gathering attribute is set, regardless of the configured value of the nG attribute.
- The Reordering attribute is set, regardless of the configured value of the nR attribute.
- The Early Acknowledgement attribute is set, regardless of the configured value of the nE attribute.

Whether or not attributes are classified as mismatched is determined strictly by the memory attributes derived from the page-table entry.

B2.7.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit memory read effects to any two adjacent bytes in memory, *p* and *p+1*, generated by the same instruction, and for two explicit memory write effects to any two adjacent bytes in memory, *p* and *p+1*, that are generated by the same instruction:
  - — The bytes *p* and *p+1* must have the same memory type and Shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LD1`, `ST1`, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - — Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes *p* and *p+1*.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see *Crossing a page boundary with different memory types or Shareability attributes* on page K1-11247.

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule, then behavior is CONSTRAINED UNPREDICTABLE, and *Crossing a peripheral boundary with a Device access* on page K1-11248 describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

  Note

  - — The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.
  - — This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.
  - — Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

B2.8 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.
- When FEAT_XS is implemented, XS attribute.

Collectively these are referred to as memory attributes.

If FEAT_MTE2 is implemented, accesses to a location which use a common definition of the memory attributes but the Tagged attribute of that location differs do not cause a mismatched access to occur.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of *Speculative* read accesses.
- Prohibition on Gathering.
- Prohibition on reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - In the EL1&0 translation regime, HCR_EL2.MIOCNCE has a value of 0.
     - All aliases with write permission have the Inner Cacheability attribute the same as the Outer Cacheability attribute.

- Either:
  - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.
  - All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.
   - Inner Non-cacheable, Outer Non-cacheable Normal memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   - Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a `DMB` barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see *Ordering and completion of data and instruction cache instructions* on page D4-4751.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
    - Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
    - A `DMB` barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

    Note

    The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Note

As described in *Non-cacheable accesses and instruction caches* on page D4-4737, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.

B2.9 Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table B2-3 on page B2-208 shows the synchronization primitives and the associated `CLREX` instruction.

Table B2-3 Synchronization primitives and associated instruction, A64 instruction set

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	`LDXRB`	`STXRB`	-
	Load-Acquire/Store-Release	`LDAXRB`	`STLXRB`	-
Halfword	-	`LDXRH`	`STXRH`	-
	Load-Acquire/Store-Release	`LDAXRH`	`STLXRH`	-
Register[b]	-	`LDXR`	`STXR`	-
	Load-Acquire/Store-Release	`LDAXR`	`STLXR`	-
Pair[b]	-	`LDXP`	`STXP`	-
	Load-Acquire/Store-Release	`LDAXP`	`STLXP`	-
None	Clear-Exclusive	-	-	`CLREX`

a. Instruction in the A64 instruction set.
b. A register instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register
A pair instruction operates on two doublewords if access X registers, or on two words if accessing W registers.

Except for the row showing the `CLREX` instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see *Marking and the size of the marked memory block* on page B2-214. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- *Exclusive access instructions and Non-shareable memory locations* on page B2-209.
- *Exclusive access instructions and Shareable memory locations* on page B2-210.
- *Marking and the size of the marked memory block* on page B2-214.
- *Context switch support* on page B2-214.
- *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-215.

- *Use of WFE and SEV instructions by spin-locks* on page B2-218.

B2.9.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place, the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state, the monitor is *set*.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Open Access state, the monitor is *clear*.

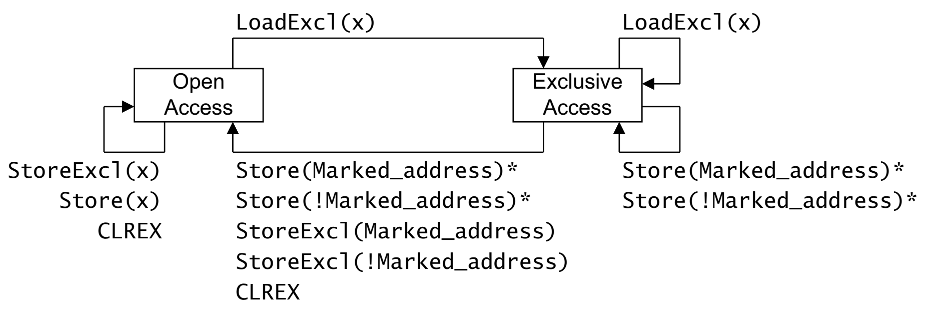
The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure B2-4 on page B2-210 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-4 Local monitor state machine diagram

For more information about marking, see *Marking and the size of the marked memory block* on page B2-214.

Note

For the local monitor state machine, as shown in Figure B2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B2-4 must not indefinitely delay forward progress of execution.

B2.9.2 Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory locations* on page B2-209, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  - — Updating memory.
  - — Returning a status value of 0.

  The local monitor can ignore accesses from other PEs in the system.
- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  - — Can hold at least one marked block.
  - — Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, see *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-215.

Note

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

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

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note

If FEAT_MTE2 is implemented, it is IMPLEMENTATION DEFINED whether a global monitor monitors access to the Tag PA space. For more information, see Chapter D6 *The Memory Tagging Extension*.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all Arm PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

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

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

The architecture only requires that *Conventional memory* mapped in this way supports this functionality.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the EL1&0 translation regime then:

  — If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.

  — Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by Arm PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:
- Some address ranges.
- Some memory types.

Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor supports only a single outstanding exclusive access to shareable memory per PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:

  — A status value of 0 is returned to a register to acknowledge the successful store.

  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.

  — If the address accessed is marked for exclusive access in the global monitor state machine for any other PE, then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:

  — A status value of 1 is returned to a register to indicate that the store failed.

  — The global monitor is not affected and remains in Open Access state for the requesting PE.

- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - — If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  - — If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means that it responds to:

- Accesses generated by PE(n).
- Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

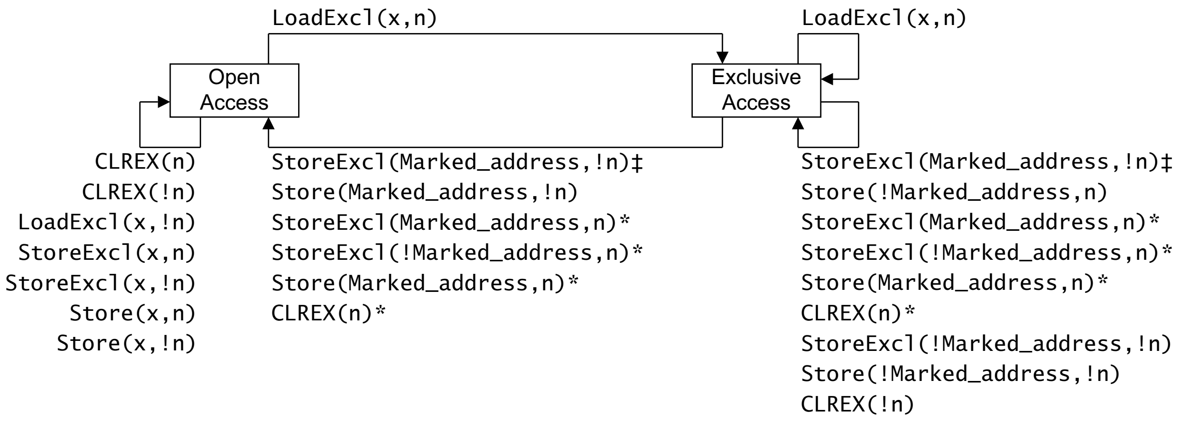
A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see *Mechanisms for entering a low-power state* on page D1-4639.

Figure B2-5 shows the state machine for PE(n) in a global monitor.



‡`StoreExcl`(Marked_address,!n) clears the monitor only if the `StoreExcl` updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see *Marking and the size of the marked memory block* on page B2-214.

Note

For the global monitor state machine, as shown in Figure B2-5 on page B2-213:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B2-5 on page B2-213 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

B2.9.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4-512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where a is 4.
- 512 words in an implementation where a is 11.

For example, in an implementation where a is 4, a successful `LDXRB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B2.9.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

B2.9.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDXP`/`STXP` pair or a `LDXR`/`STXR` pair. To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a `LoadExcl`/`StoreExcl` instruction pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B2-3 on page B2-208. In this context, a `LoadExcl`/`StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - — The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched addresses.

  - — The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with mismatched transaction sizes.

  - — The block of data of the size of the larger of the transaction sizes used by the `LoadExcl`/`StoreExcl` pair at the address accessed by the `LoadExcl`/`StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.
- An implementation of the `LoadExcl` and `StoreExcl` instructions can require that, in any thread of execution, the `StoreExcl` instruction accesses the same number of registers as the preceding `LoadExcl` instruction executed in that thread. If the `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they access the same number of registers. For more information, see *CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers* on page B2-217.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the Tag Checked property of a memory access due to a `StoreExcl` instruction is the same as the Tag Checked property of a memory access by the preceding `LoadExcl` instruction executed in that thread. If the Tag Checked property of memory accesses due to a `LoadExcl`/`StoreExcl` pair in the same thread of execution differ, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of such a `LoadExcl`/`StoreExcl` pair, and fail for other instances of such a `LoadExcl`/`StoreExcl` pair.

  - — The data at the address accessed by the LoadExcl/StoreExcl pair is UNKNOWN.

This means software can rely on a `LoadExcl`/`StoreExcl` pair to eventually succeed only if the memory is accessed with the same Tag Checked property.

- `LoadExcl`/`StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl`/`StoreExcl` loop within a single thread of execution, the software meets all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, preloads, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, `ISB` barriers, or indirect branches.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:
  - There are no stores or `PRFM` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive.
  - There are no loads or preloads to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.
  - There are no direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.
  - All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

  The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

  — Performing stores to a PA covered by the Exclusives monitor.
  — Prefetching with intent to write to a PA covered by the Exclusives monitor.
  — Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.
  — Executing instruction cache invalidate all instructions.
  — Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.
  — Executing TLB maintenance to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.
- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.
- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.
- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:

  — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

  Note

  This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.

  — The data at the address accessed by the `StoreExcl` is UNKNOWN.

Note

Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `STREX` instruction, and the change is performed using a break-before-make sequence as described in *Using break-before-make when updating translation table entries* on page D5-4913, if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

  Note

  — The TLB invalidation will clear either the local or global monitor.

  — The PA will be checked between the `LoadExcl` and `StoreExcl`.

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PFRM PST*` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PFRM PST*` accesses to a memory location might impede the forward progress of another PE.
- If FEAT_MTE2 is implemented, and if a Tag Unchecked store exclusive instruction would not perform the store and return a status value of 1, it is CONSTRAINED UNPREDICTABLE whether:
  — The instruction is a Tag Checked access,

  — The instruction is an Tag Unchecked access.

  For more information, see Chapter D6 *The Memory Tagging Extension*.

Note

In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers

As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED UNPREDICTABLE if, in a single thread of execution, either:

- An `LDXP` instruction of two 32-bit quantities is followed by an `STXR` instruction of one 64-bit quantity at the same address.
- An `LDXR` instruction of one 64-bit quantity is followed by an `STXP` instruction of two 32-bit quantities at the same address.

In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:

- The `STXP` or `STXR` instruction generates an external Data Abort.
- The `STXP` or `STXR` instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `0b110101`.
- The `STXP` or `STXR` instruction always fails, returning a status of 1.
- The `STXP` or `STXR` instruction always passes, returning a status of 0.
- This `STXP` or `STXR` instruction has the same pass or fail behavior that it would have had if the instruction had used the same size and number of registers as the preceding `LDXR` or `LDXP` instruction.

B2.9.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, and `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in *Wait for Event* on page D1-4639. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an `SEVL` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.